University Bremerhaven

Course Documentation

Embeded Systems Project (LAB)
[ES–PRO]

- Part 1: System analysis and planing
- Part 2: Modeling and simulation
- Part 3: Interface specification
- Part 4: Hardware/Software specification
- Part 5: Embedded Controller design
- Part 6: Algorithm development
- Part 7: Industrial/medical applicationMedical Devices and Applications

Revision: V1.2a

Release: October 2020

Prof. Dr.-Ing. Kai Mueller
University of Applied Sciences Bremerhaven
Institute for Automation and Electrical Engineering
An der Karlstadt 8

D–27568 Bremerhaven / Germany

Phone: +49 471 48 23 – 415
FAX: +49 471 48 23 – 555

Email: kmueller@hs–bremerhaven.de

I Introduction

I.I Course Documentation

See <http://www1.hs–bremerhaven.de/kmueller/> for updates.

I.II Embedded Systems Project (lab)

The embedded design project teaches how to design complete embedded systems and how to write technical documentation. More the 14 billion embedded systems world wide became part our daily life. Malfunctions of devices especially in medical and industrial environment may expose people to hazards. This should be considered in the design. Bringing these devices to market requires to pay attention to these issues.

The projects labs focus on the design of embedded systems in hardware and in software. It will show the transition from development to production systems.

Bremerhaven, September 2014

Kai Müller
<kmueller@hs–bremerhaven.de>
phone: +49 471 4823 – 415

II Contents

1 Processes for the Embedded Systems Project 1

1.1 Inverted Pendulum Control System 1

1.2 Pulse Oximeter 1

1.3 Magnetic Levitation 2

1.4 Processor Boot Procedure 3

1.4.1 SREC Format 3

1.4.2 MCS (Intel HEX) File Format 5

Lab #01a: 7

1.5 LAB Experiment: Large Application Boot Procedure 7

1.5.1 PicoBlaze Parallel Flash Programmer/Reader 8

1.5.2 MicroBlaze Boot Application 14

Lab #02:
Boot Application with EDK/SDK 18

1.6 Creating a Bootloader with EDK 18

1.6.1 Bootloader and Configuration Extensions 20

Lab #01b: 22

1.7 LAB Experiment: Vivado Embedded System Development and Boot Procedure 22

Lab #01c: 46

1.8 LAB Experiment: Zynq Interrupt 46

Interface Hardware 52

2 Pulse Oximeter Interface Hardware 52

2.1 Finger Probe 52

2.2 Interface Block Diagram 52

3 Magnetic Levitation Model 60

4 Electrical Interface to the Cart-Pendulum Experiment 65

4.1 I/O Signals 71

4.1.1 Data Formats 71

4.2 Process User Interface 72

4.2.1 Internal Commands 73

4.2.2 External Commands 73

4.3 Retrieving Process Data 74

5 PI Controller for the Initialization Phase 76

5.1 Fixed Point PI Controller 78

6 C Code Verification in Simulink 81

7 Identification of Process Subsystems 86

7.1 Subsystem Modelling 86

7.2 ARX Parameter Identification 89

7.2.1 Numerical Solution of the Estimation Problem 92

Lab #03: 95

7.3 Identification of the System from Section 7.1 95

7.4 Recursive Identification (RLS) 97

7.4.1 Extended Model Identification (Disturbance Model) 98

Lab #04: 101

7.5 Recursive Identification of the System from Section 7.1 101

8 Bibliography 102

1 Processes for the Embedded Systems Project

The Embedded Systems Project should provide the transition from theory and design/development to production systems.

1.1 Inverted Pendulum Control System

The practical part of the Embedded Design Project involves the design and implementation of a SoC for the inverted/non-inverted pendulum process.

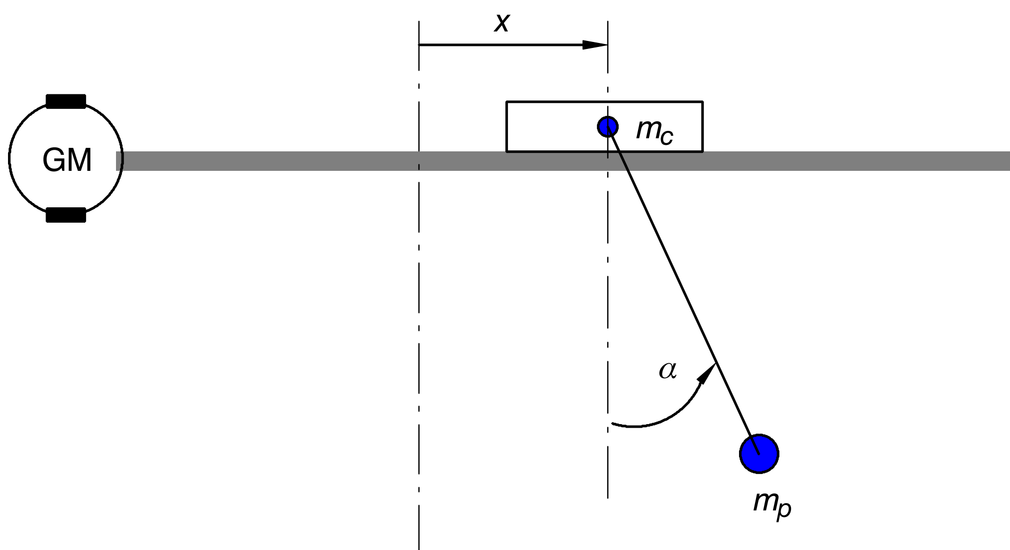


Figure 1.1: Cart pendulum process

The process has two degrees of freedom (α, x) and therefore is a fourth order system. Modeling, simulation and controller design is not considered in this lab. Subject of this lab are the control system setup, controller implementation, user interface and security considerations.

1.2 Pulse Oximeter

A pulse oximeter is a medical embedded system for measuring oxygen saturation in blood and an “optical” ECG as well as the heart beat rate.

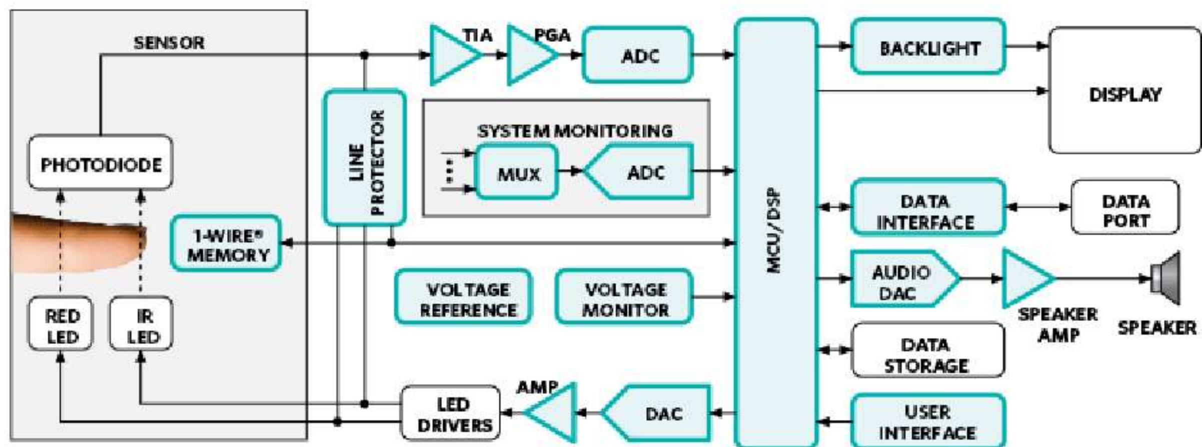


Figure 1.2: Pulse oximeter block diagram (© Maxim Integrated)

1.3 Magnetic Levitation

Magnetic levitation system are used for friction free bearings and magnetically levitated vehicles.

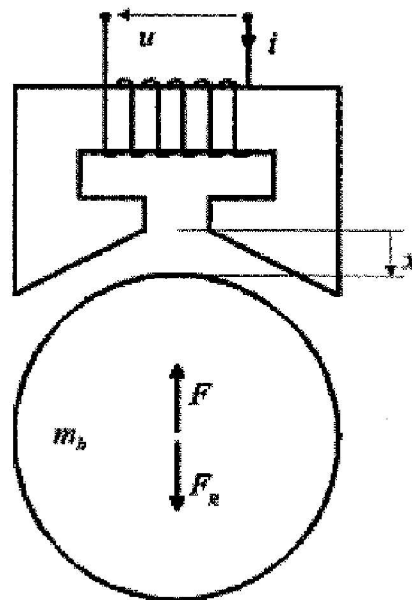


Figure 1.3: Magnetic levitation system with ferromagnetic ball

Figure 1.4: Magnetic levitation application

1.4 Processor Boot Procedure

Any embedded system must configure (boot) itself on power-on condition. In previous labs the software was loaded via the development system by using the mdm (microprocessor debug module). The FPGA configuration and the software must be stored in flash memory, This is done either in platform flash, SPI flash, NOR-flash or NAND-flash depending on device type and memory size considerations. While programs can be executed directly from flash it is often much faster to copy the flash contents to DRAMs. This is called the boot process. This procedure takes several milliseconds until a device is ready to perform the required functions.

The Spartan-3e prototype board ha s 128MB Intel NOR-flash memory large enough to store any big application software. This “main” application mus be transferred to DRAM by a very small boot application which must reside in internal block memory (e.g. limited to some kbyte of code).

Flash or PROM contents are described by a hex file format. There are two popular formats for this purposes: SREC (Motorola s-record format) and MCS (Intel hex file format).

1.4.1 SREC Format

An SREC format file consists of a series of ASCII records. All hexadecimal (hex) numbers are Big Endian. The records have the following structure:

1. **Start code**, one character, an S.
2. **Record type**, one digit, 0 to 9, defining the type of the data field.
3. **Byte count**, two hex digits, indicating the number of bytes (hex digit pairs) that follow in the rest of the record (in the address, data and checksum fields).
4. **Address**, four, six, or eight hex digits as determined by the record type for the memory location of the first data byte. The address bytes are arranged in big endian format.
5. **Data**, a sequence of 2n hex digits, for n bytes of the data. 5.

6. **Checksum**, two hex digits – the least significant byte of ones' complement of the sum of the values represented by the two hex digit pairs for the byte count, address and data fields.

Example

```
S00F000068656C6C6F202020202000003C
S11F00007C0802A6900100049421FFF07C6C1B787C8C23783C6000003863000026
S11F001C4BFFFFE5398000007D83637880010014382100107C0803A64E800020E9
S111003848656C6C6F20776F726C642E0A0042
S5030003F9
S9030000FC
```

Explanation of record types

S0 The S0 record data sequence contains vendor specific data rather than program data. String with file name and possibly version info.

S1, S2, S3 Data sequence, depending on size of address needed. A 16–bit/64K system uses S1, 24–bit address uses S2 and full 32–bit uses S3.

S5 Count of S1, S2 and S3 records previously appearing in the file or transmission. The record count is stored in the 2–byte address field. There is no data associated with this record type.

S7, S8, S9 The address field of the S7, S8, or S9 records may contain a starting address for the program.

Record	Description	Address Bytes	Data Sequence
S0	Block header	2	Yes
S1	Data sequence	2	Yes
S2	Data sequence	3	Yes
S3	Data sequence	4	Yes
S5	Record count	2	No
S7	End of block	4	No
S8	End of block	3	No
S9	End of block	2	No

Figure 1.5: SREC format record summary (Wikipedia)

1.4.2 MCS (Intel HEX) File Format

Another commonly used format is the little endian equivalent MCS. Like SREC this format is used by many manufacturers.

The Intel Hexadecimal Object record format has a 9-character (4 field) prefix that defines the start of the record, byte count, load address, and record type, as well as a 2-character checksum suffix.

The 16-bit hexadecimal format allows for a 20-bit segmented address space, and the 32-bit format allows for the 32-bit linear address space.

The six record types are:
00 = **Data Record**
01 = **End of File Record** (signals the end of the file)
02 = **Extended Segment Address Record** (provides the offset to determine the absolute destination address)
03 = **Start Segment** Address Record (is ignored during input and not sent during output by Data I/O translator firmware)
04 = **Extended Linear Address Record** (provides the offset to determine the absolute destination address) *
05 = **Start Linear Address Record** (provides the execution start address)
The Extended Linear Address Record type and Start Linear Address Record type are supported only in the 32–bit format.

Example

```
:020000021000EC
:10010000214601360121470136007EFE09D2190140
:100110002146017EB7C20001FF5F16002148011988
:10012000194E79234623965778239EDA3F01B2CAA7
:100130003F0156702B5E712B722B732146013421C7
:00000001FF
```

Input Data Record (Type 00)

:	*BC*	AAAA	00	*HHHH....HH*	CC
Start Char	Byte Count	Hex Address	Record Type	HH = 1 data byte	Checksum
	2 chars	4 chars	2 chars	2 up to 32 chars	2 chars

End of File Record (Type 01)

:	00	0000	01	FF
Start Char	Byte Count	Hex Address	Record Type	Checksum
	2 chars	4 chars	2 chars	2 chars

Extended Segment Address Record (Type 02)

:	*BC*	0000	02	*HHHH*	CC
Start Char	Byte Count	Hex Address	Record Type	2 Byte Offset	Checksum
	2 chars	4 chars	2 chars	4 chars	2 chars

Extended Linear Address Record (Type 04)

:	02	0000	04	*HHHH*	CC
Start Char	Byte Count	Hex Address	Record Type	2 Byte offset	Checksum
	2 chars	4 chars	2 chars	2 up to 32 chars	2 chars

Figure 1.6: MCS record summary (Xilinx)

For flash programming it is sometimes required to convert one format to the other. From the programmer's point of view both formats save the same purpose.

Lab #01a:

1.5 LAB Experiment: Large Application Boot Procedure

For the lab we create an application with is too big to fit into internal BRAM. The application software is stored in NOR-flash memory. A small boot loader will copy the contents of the flash memory into DRAM and transfers execution to this application.

The operation and programming of the parallel “Strataflash”memory can be easily verified by a PicoBlaze application. This design will not be explained, instead, it will only be used to verify programming.

For programming the parallel flash memory the configuration bits of the FPGA need to be changed to a position where the platform flash memory is disabled. Otherwise the bit D0 (data bit 0) of the NOR-flash conflicts with the output of the platform flash memory for normal configuration of the FPGA.

	Master Serial	**SPI**	**BPI**	**Slave Parallel**	**Slave Serial**	**JTAG**
M[2:0] mode pin settings	<0:0:0>	<0:0:1>	<0:1:0>=Up <0:1:1>=Down	<1:1:0>	<1:1:1>	<1:0:1>
Data width	Serial	Serial	Byte-wide	Byte-wide	Serial	Serial
Configuration memory source	Xilinx Platform Flash	Industry-standard SPI serial Flash	Industry-standard parallel NOR Flash or Xilinx parallel Platform Flash	Any source via microcontroller, CPU, Xilinx parallel Platform Flash, etc.	Any source via microcontroller, CPU, Xilinx Platform Flash, etc.	Any source via microcontroller, CPU, System ACE™ CF, etc.
Clock source	Internal oscillator	Internal oscillator	Internal oscillator	External clock on CCLK pin	External clock on CCLK pin	External clock on TCK pin
Total I/O pins borrowed during configuration	8	13	46	21	8	0
Configuration mode for downstream daisy-chained FPGAs	Slave Serial	Slave Serial	Slave Parallel	Slave Parallel or Memory Mapped	Slave Serial	JTAG
Stand-alone FPGA applications (no external download host)	✓	✓	✓	Possible using XCFxxP Platform Flash, which optionally generates CCLK	Possible using XCFxxP Platform Flash, which optionally generates CCLK	
Uses low-cost, industry-standard Flash		✓	✓			
Supports optional MultiBoot, multi-configuration mode			✓			

Figure 1.7: FPGA configuration modes

The figure 1.7 lists the most common configuration modes according to the M2, M1, and M0 FPGA configuration bits. The FPGA will configure automatically after power-on from the appropriate source. For this project we will select mode bits

M0 = '1' (jumper M0 open),
M1 = '0' (jumper M1 closed),
M2 = '0' (jumper M2 closed).

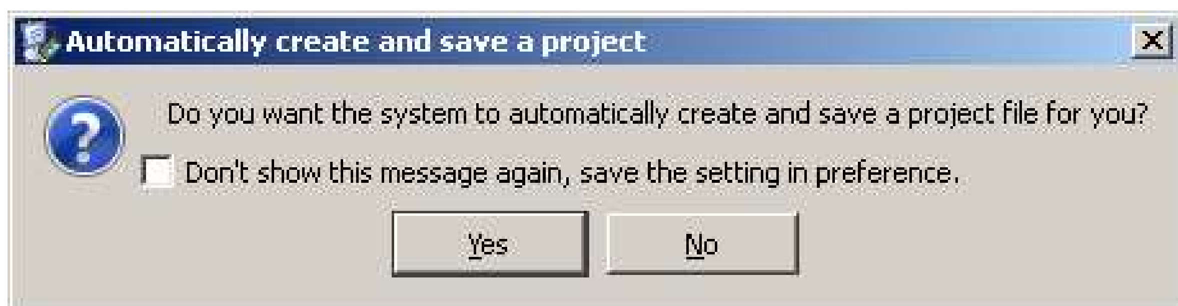
This selects BPI (**b**yte **p**eripheral **i**nterface) and thus disables the platform flash memory to avoid data contention on D0.

In this configuration mode the FPGA will not configure itself. The configuration has to be loaded by the JTAG interface which is always available if the board allows JTAG operation. After programming the NOR-flash the jumper M0 can be closed to re-enable self configuration via serial platform flash memory.

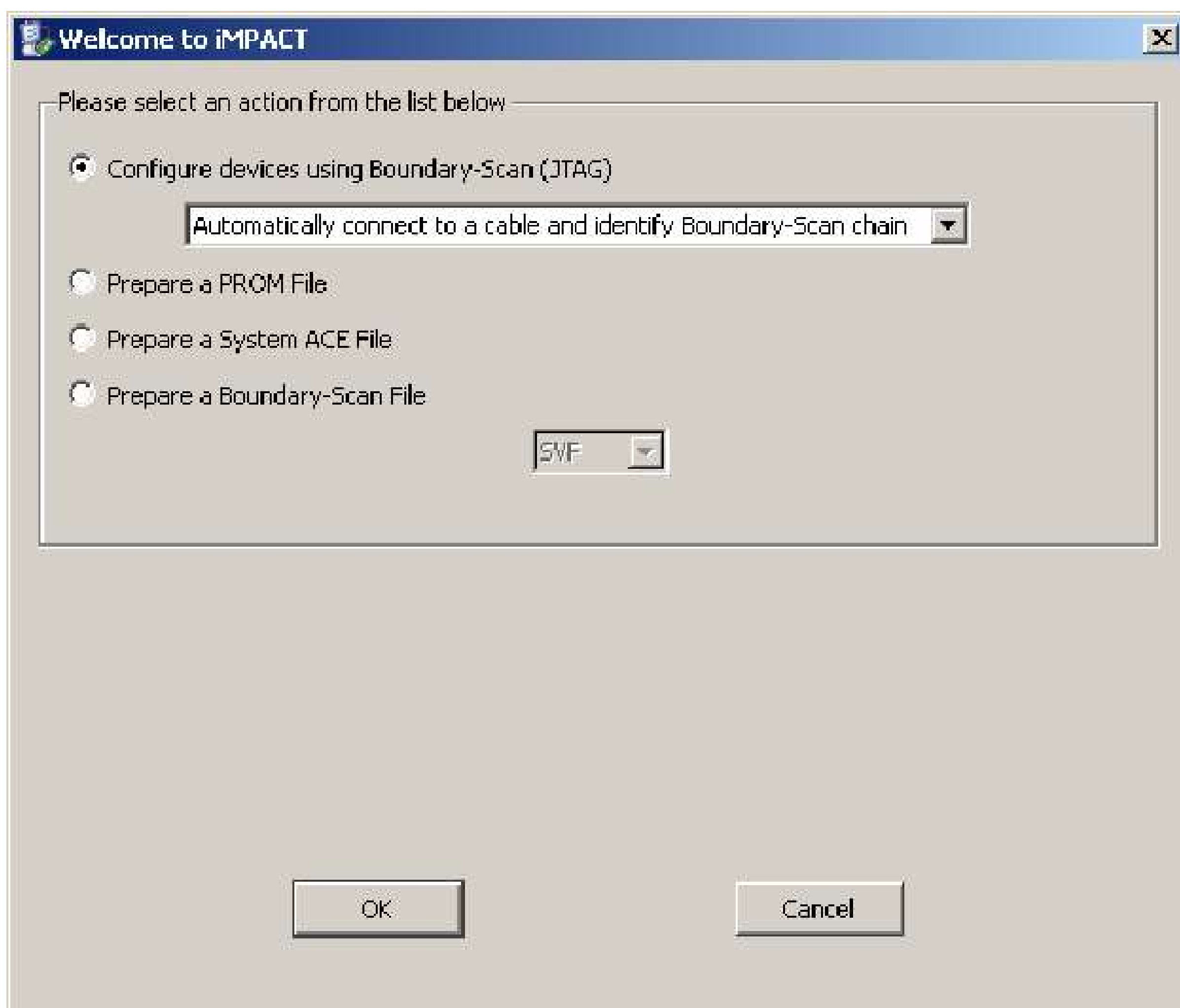
1.5.1 PicoBlaze Parallel Flash Programmer/Reader

A ready developed application can be downloaded to the FPGA by the Impact programmer. The ISE, EDK, or SDK tools are not required. The applications requires a serial connection (115200 bits/s!) and a terminal program (Termite or Putty).

- Attach the serial cable and select 115200 bits/s Baud Rate
- Start the Impact programmer



Select Yes



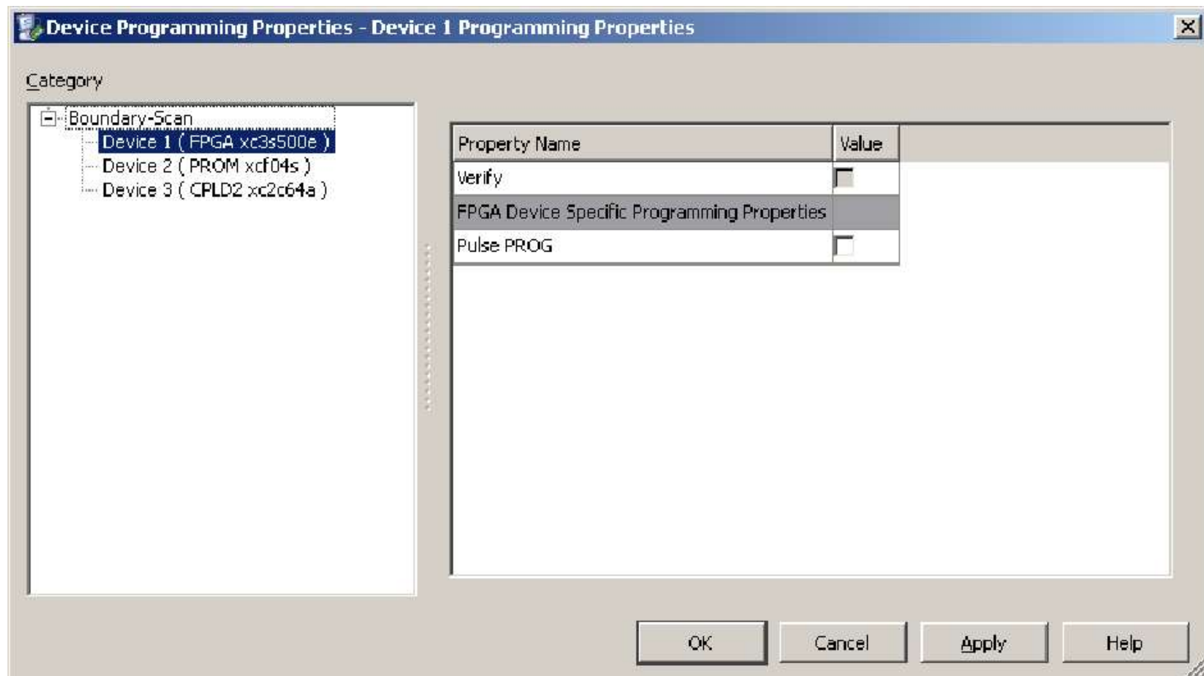
Select Boundary-Scan and click OK

In the Auto Assign Configuration File dialog click Yes

Choose the file “parallel_flash_memory_uart_programmer.bit” and click Open

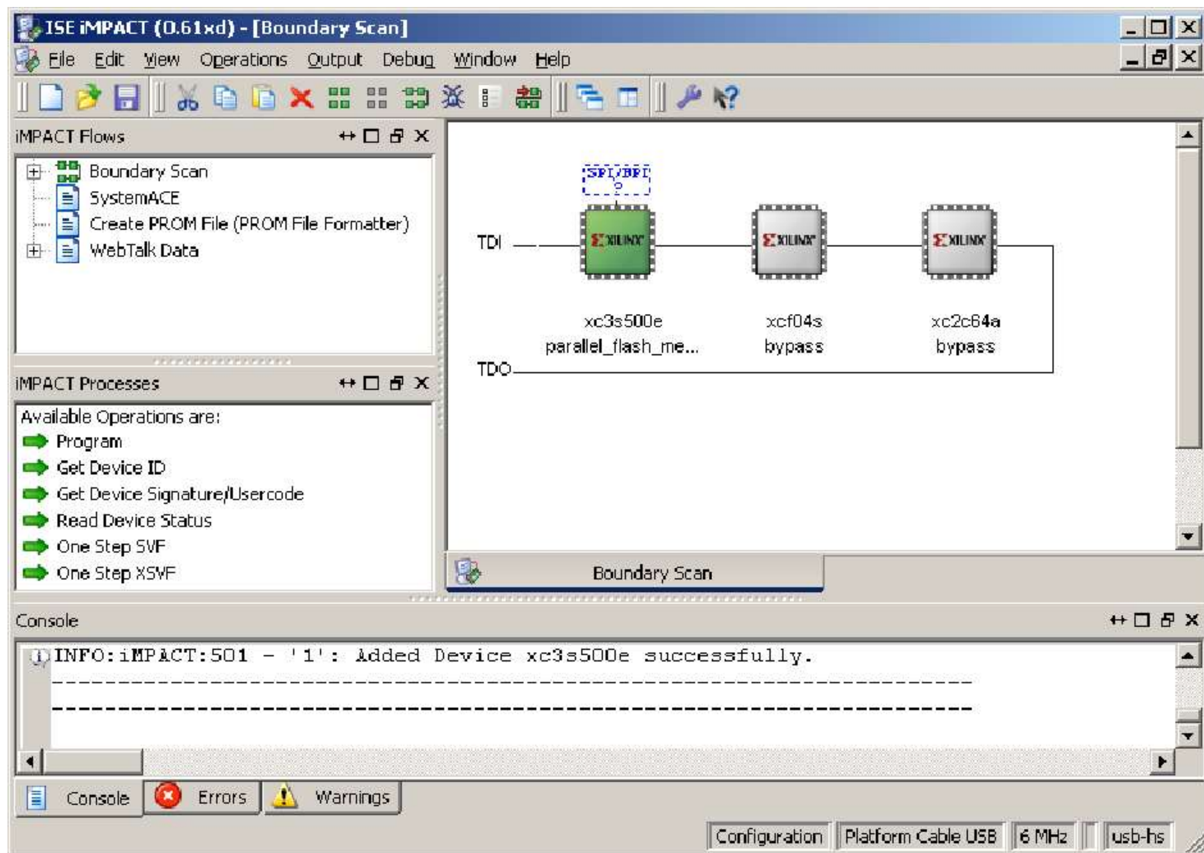
In the Attach SPI or BPI ROM dialog select No.
Click on Bypass in the Assign New Configuration File dialog.

Do the same thing with the CoolRunner™ xc2c64a CPLD device (Bypass)

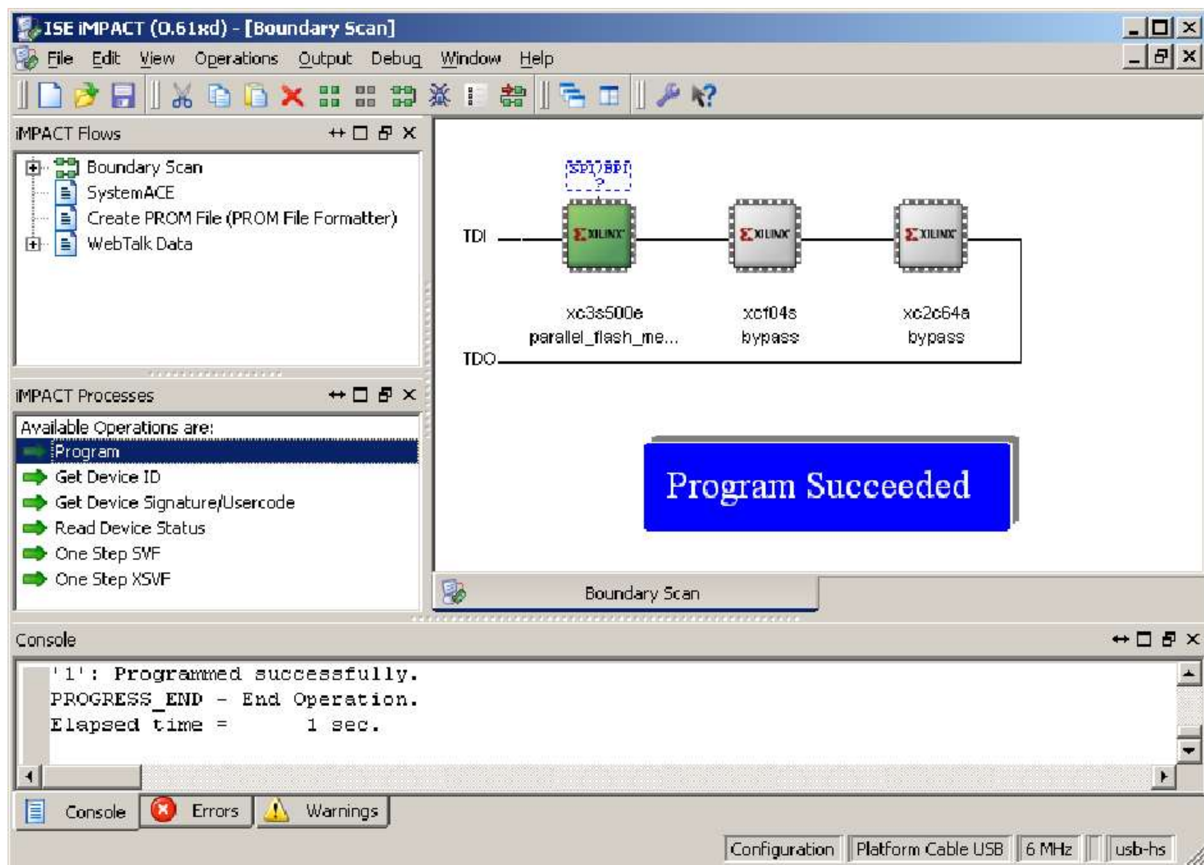


Click on OK in the Device Programming Properties dialog.

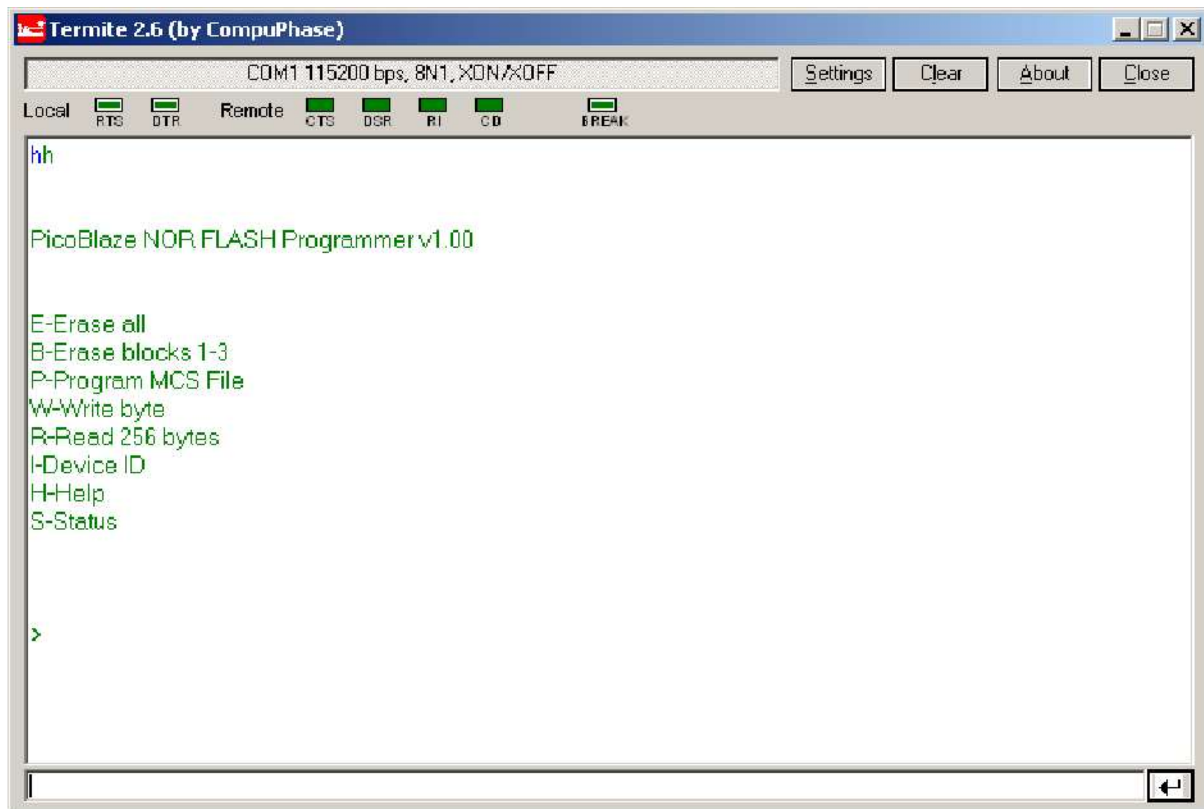
- Select the FPGA in Impact and double-click on the Program option (left).



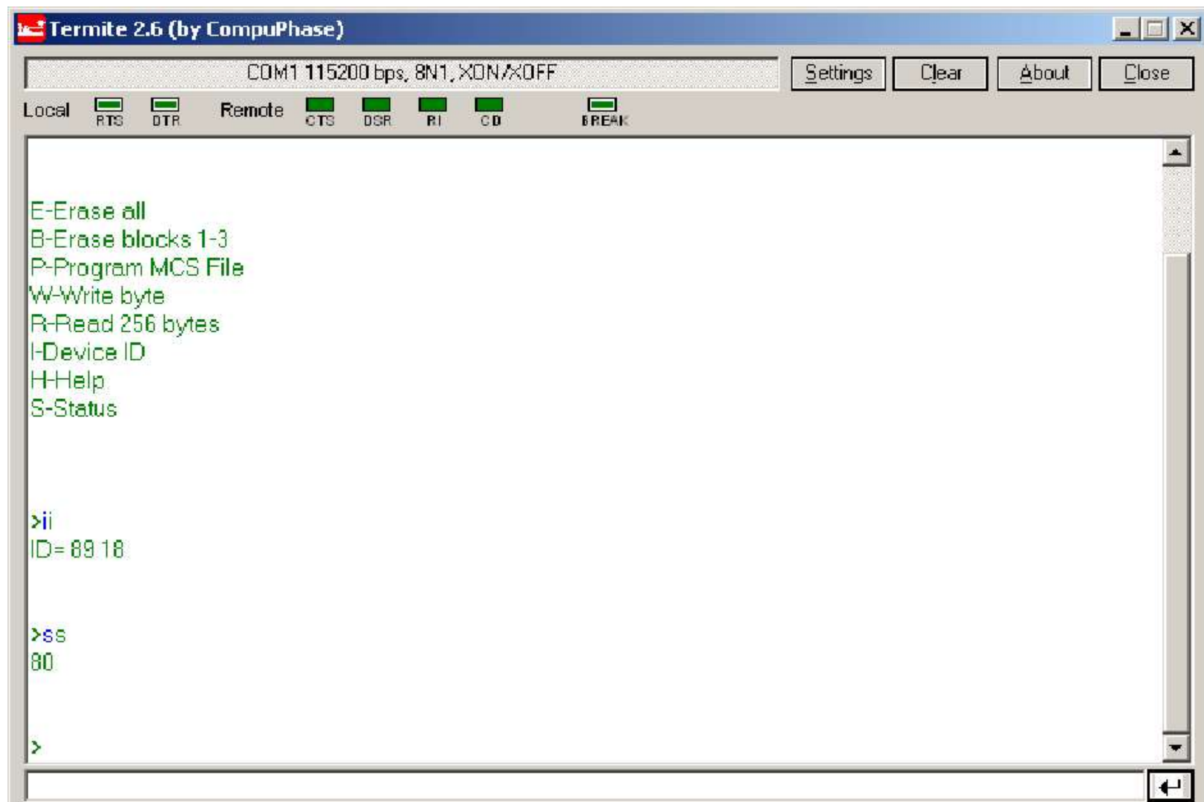
You should see the following result.



▶ You should see the start-up screen of the programmer on the terminal program



▶ Enter I (Flash ID code) command and the S (status) command. The normal status code must be 0x80.



- Inspect the first page (on address 0x000000) of the flash memory (might look different on your board).

Termite 2.6 (by CompuPhase)

COM1 115200 bps, 8N1, XON/XOFF | Settings | Clear | About | Close

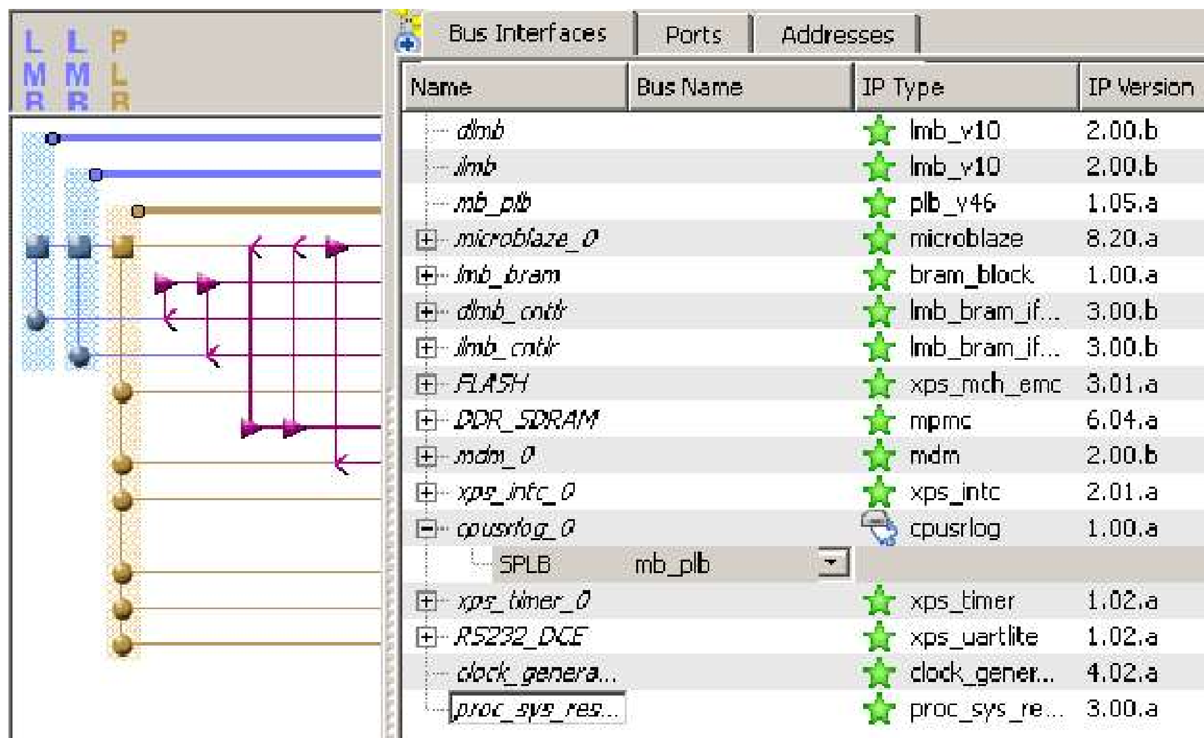
Local RTS DTR Remote CTS DSR RI CD BREAK

```
>rr
address=000000000000

000000 53 30 32 42 30 30 30 30 34 33 33 41 32 46 34 44
000010 37 35 36 35 32 46 34 35 33 31 33 33 35 30 34 43
000020 34 32 32 46 36 33 37 30 36 33 36 46 36 45 37 34
000030 37 32 32 46 37 37 36 46 37 32 36 42 37 33 37 30
000040 36 31 36 33 36 35 32 46 36 33 37 30 36 33 36 46
000050 36 45 37 34 37 32 35 46 39 45 0D 0A 53 33 30 44
000060 30 30 30 30 30 30 30 30 42 30 30 30 34 34 30 30
000070 42 38 30 38 30 30 30 30 33 45 0D 0A 53 33 30 44
000080 30 30 30 30 30 30 30 38 42 30 30 30 34 34 30 30
000090 42 38 30 38 30 39 37 34 42 39 0D 0A 53 33 30 44
0000A0 30 30 30 30 30 30 31 30 42 30 30 30 34 34 30 30
0000B0 42 38 30 38 30 39 43 30 36 35 0D 0A 53 33 30 44
0000C0 30 30 30 30 30 30 32 30 42 30 30 30 34 34 30 30
0000D0 42 38 30 38 30 39 38 43 38 39 0D 0A 53 33 31 35
0000E0 34 34 30 30 30 30 30 30 42 30 30 30 34 34 30 30
0000F0 33 31 41 30 33 33 30 38 42 30 30 30 34 34 30 30

OK
```

- Do the same operation *after* programming the flash in the next section.

1.5.2 MicroBlaze Boot Application

- In EDK create a new system “cpcontr”
- Using the “Base System Builder” configure the system with UART (DCE, 115200 bits/s, timer (interrupt), and parallel flash support according to the following figure.



- Create a user logic peripheral “cpusrlog” (2 registers) with the following external signals

```
--USER ports added here
       cup_SWITCH : in std_logic_vector(3 downto 0);
       cup_LED            : out std_logic_vector(7 downto 0);
       cup_SPI_SS_B : out std_logic;
       cup_AMP_CS : out std_logic;
       cup_AD_CONV : out std_logic;
       cup_FPBA_INIT_B : out std_logic;
       cup_LCD_rw : out std_logic;
       cup_LCD_e : out std_logic;
-- ADD USER PORTS ABOVE THIS LINE ------------------
```

- Add user user_logic.vhd the following lines. This prevents other components to access the same signals that the NOR-flash is using. It also allows to access LEDs and SWITCHes.

```
  --USER logic implementation added here
         cus_LED <= slv_reg0(24 to 31);
         cus_SPI_SS_B <= ´1´;
         cus_AMP_CS <= ´1´;
         cus_AD_CONV <= ´0´;
         cus_FPBA_INIT_B <= ´0´;
         cus_LCD_rw <= ´0´;
         cus_LCD_e <= ´0´;
o
o
o
    case slv_reg_read_sel is
```

```
        when "10" =>
            slv_ip2bus_data <= slv_reg0(0 to 23) & "0000" & cus_SWITCH;
        when "01" => slv_ip2bus_data <= slv_reg1;
        when others => slv_ip2bus_data <= (others => '0');
    end case;
```

- Add the following lines to the constraints file "system.ucf" to define IO pins for the user logic:

```
### user logic constraints
NET cpusrlog_0_cup_SWITCH_pin<3> LOC = "N17" | IOSTANDARD = LVTTL | PULLUP;
NET cpusrlog_0_cup_SWITCH_pin<2> LOC = "H18" | IOSTANDARD = LVTTL | PULLUP;
NET cpusrlog_0_cup_SWITCH_pin<1> LOC = "L14" | IOSTANDARD = LVTTL | PULLUP;
NET cpusrlog_0_cup_SWITCH_pin<0> LOC = "L13" | IOSTANDARD = LVTTL | PULLUP;
NET cpusrlog_0_cup_LED_pin<7> LOC = "F9" | IOSTANDARD = LVTTL | SLEW = SLOW  | DRIVE = 8;
NET cpusrlog_0_cup_LED_pin<6> LOC = "E9" | IOSTANDARD = LVTTL | SLEW = SLOW  | DRIVE = 8;
NET cpusrlog_0_cup_LED_pin<5> LOC = "D11" | IOSTANDARD = LVTTL | SLEW = SLOW  | DRIVE = 8;
NET cpusrlog_0_cup_LED_pin<4> LOC = "C11" | IOSTANDARD = LVTTL | SLEW = SLOW  | DRIVE = 8;
NET cpusrlog_0_cup_LED_pin<3> LOC = "F11" | IOSTANDARD = LVTTL | SLEW = SLOW  | DRIVE = 8;
NET cpusrlog_0_cup_LED_pin<2> LOC = "E11" | IOSTANDARD = LVTTL | SLEW = SLOW  | DRIVE = 8;
NET cpusrlog_0_cup_LED_pin<1> LOC = "E12" | IOSTANDARD = LVTTL | SLEW = SLOW  | DRIVE = 8;
NET cpusrlog_0_cup_LED_pin<0> LOC = "F12" | IOSTANDARD = LVTTL | SLEW = SLOW  | DRIVE = 8;
NET cpusrlog_0_cup_SPI_SS_B_pin LOC = "U3" | IOSTANDARD = LVCMOS33  | SLEW = SLOW  | DRIVE =
6;
NET cpusrlog_0_cup_AMP_CS_pin LOC = "N7" | IOSTANDARD = LVCMOS33 | SLEW = SLOW  | DRIVE = 6;
NET cpusrlog_0_cup_AD_CONV_pin LOC = "P11" | IOSTANDARD = LVCMOS33 | SLEW = SLOW  | DRIVE =
6;
NET cpusrlog_0_cup_FPBA_INIT_B_pin LOC = "T3" | IOSTANDARD = LVCMOS33 | SLEW = SLOW  | DRIVE
= 4;
NET cpusrlog_0_cup_LCD_rw_pin  LOC = "L17" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 2;
NET cpusrlog_0_cup_LCD_e_pin  LOC = "M18" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 2;
```

- Submit Hardware => Generate Bitstream and transfer the design to SDK.
- In SDK create a "peripheral_tests" application with is to big to fit into the internal BRAM. Verify proper operation by executing this program on the Spartan-3e board.
- Add code to access LEDs and SWITCHes and verify that the hardware is accessed properly.

```
SREC Bootloader
Loading SREC image from flash @ address: 82000000

Bootloader: Processed (0x)00000001 S-records
Bootloader: Processed (0x)00000002 S-records
Bootloader: Processed (0x)00000003 S-records
Bootloader: Processed (0x)00000004 S-records
o
o
o
```

```
Bootloader: Processed (0x)0000033a S-records
Bootloader: Processed (0x)0000033b S-records
Executing program starting at address: 00000000

---Entering main---
The system has successfully booted from FLASH!

Running IntcSelfTestExample() for xps_intc_0...
IntcSelfTestExample PASSED
Intc Interrupt Setup PASSED

Running UartLiteSelfTestExample() for mdm_0...
UartLiteSelfTestExample PASSED

 Running TmrCtrSelfTestExample() for xps_timer_0...
TmrCtrSelfTestExample PASSED

 Running Interrupt Test  for xps_timer_0...
Timer Interrupt Test PASSED
---Exiting main---
```

Lab #02:

Boot Application with EDK/SDK

1.6 Creating a Bootloader with EDK

If industry standard Flash memory is available (BPI = byte parallel interface) creating a bootloader becomes easy. However, it is necessary to include Flash memory access in hardware. The Spartan-3E board has a 128 MByte parallel Flash memory which is large enough to hold FPGA configuration as well as several applications programs.

- Create a new EDK project "sp3boot" within a directory of that name.
- Include the parallel Flash memory controller "xps_mch_emc" [XPS Multi-Channel External Memory Controller (SRAM/Flash)] with the Base System Builder wizard. Verify that the FLASH instance is included in your design (see fig. 1.8).

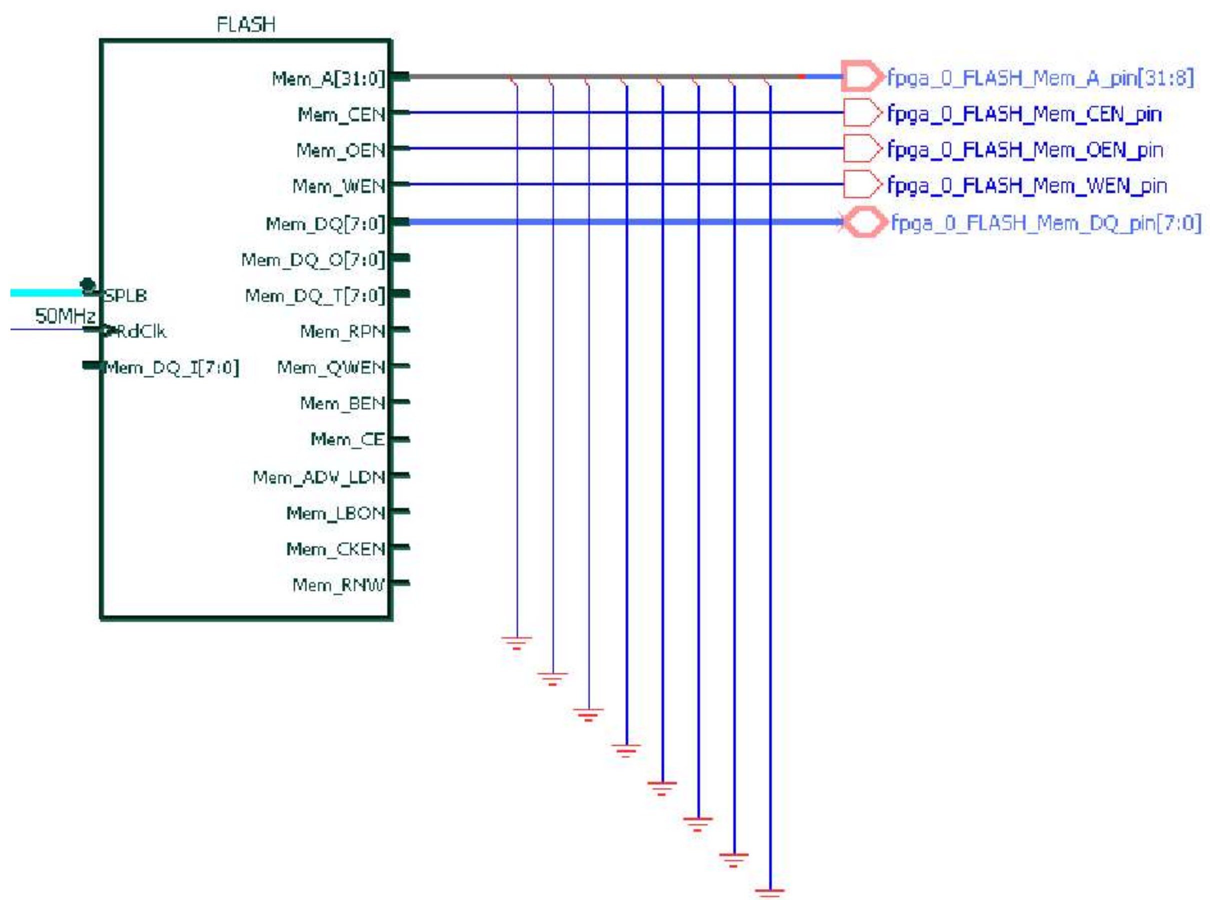


Figure 1.8: FLASH instance

- Export the design to SDK and create an application in DDR memory (i.e. a "helloworld" application and verify that this application executes in external memory properly.

- Program the parallel Flash memory with the corresponding .elf file with an offset of 0x0100000 in Flash memory. The FPGA image (.bit file) requires approximately 4Mbit of data. This corresponds to 500 KByte or 0x07A120 (i.e. Matlab: `dec2hex(4e6/8)`). So a safe offset is 0x0200000 (2 MBytes). This first bytes can then hold the FPGA configuration data and loaded by BPI up (byte parallel interface upward addressing). Be sure to select the checkbox: `Convert ELF to bootable SREC format and program`.

- Create an SREC_bootloader application. In the header file `blconfig.h` define `FLASH_IMAGE_BASEADDR` to the correct memory address (where the srec file has been programmed). Note that the SREC_bootloader executes from internal block memory. Why?

- Execute the SREC_bootloader and verify that bootloading works and your first application program is executed after the bootloader has finished.
  Explain the line of code
  ```
  (*laddr)();
  ```
  in `bootloader.c`.

- Finally the SREC_bootloader should reside in the platform flash memory so that the MicroBlaze executes it after power on. The .bit file is created by the `data2mem` utility which initializes BRAM with the SREC_bootloader.elf contents. Figure 1.9 illustrates this.

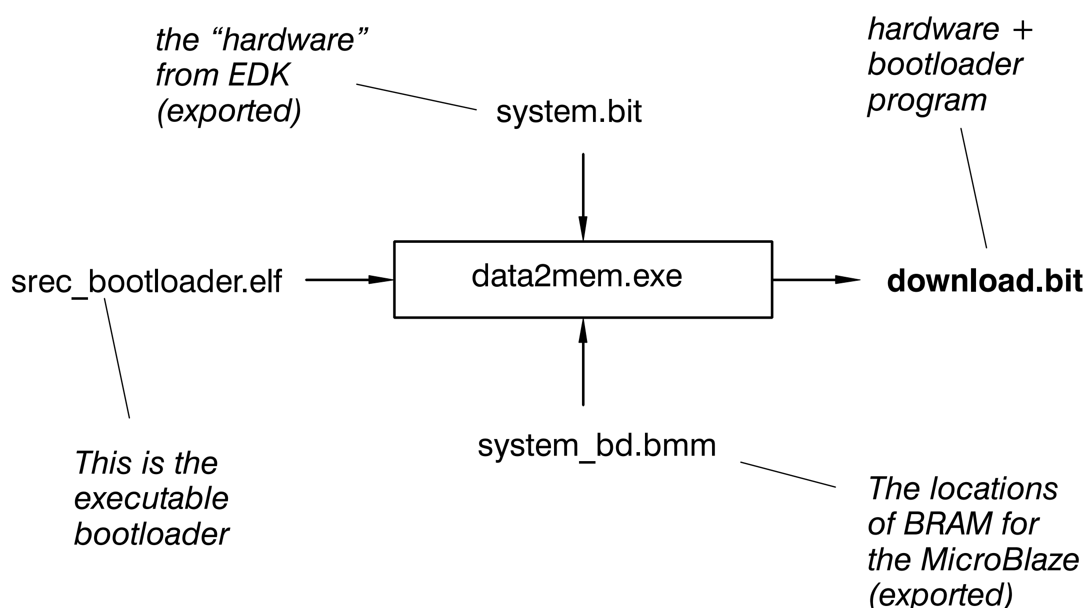


Figure 1.9: Data2Mem BRAM initialization

All configuration files are located under ...workspace sp3boot_hw_platform.

▶ The file `download.bit` will be usually programmed in the configuration flash memory. Don't do this here! Instead we will simulate the configuration process by programming the FPGA with the Impact programmer.

Copy the following files (some file names may be different according to your selections):

```
download.bit
Flashboot.elf.srec
MbBoot.elf
system.bit
system_bd.bmm
```

to another place (e.g. `\mueller\ESPRO\PlatformBoot`). Invoke the Impact programmer and verify that the bootloader is functional. Flashing the configuration memory would have exactly the same effect.

1.6.1 Bootloader and Configuration Extensions

▶ Inspect the "srec_bootloader_0.elf" software by converting it into binary form. Please copy the following files to an "Analysis" directory before performing the lab tasks.

```
data2mem -bd srec_bootloader_0.elf -d -o m code.mem
```

(The ISE Design Suite Command Prompt is required to execute this!)

Inspect the code.mem file with an ASCIII Editor (i.e. Jedit).

▶ Look at the formatted "srec_bootloader_0.elf" contents by viewing the contents of "hello_elf.dump". This file can be created by the command

```
data2mem -bd srec_bootloader_0.elf -d >hello_elf.dump
```

Compare the contents of the two files "hello_elf.dump" and "code.mem".

▶ Analyze the dump of the "download.bit" file. Since the bit file in binary it it requires to dump it into an ASCII file "bootlbit.dump".

```
data2mem -bm system_bd.bmm -bt download.bit
          -de >bootlbit.dump
```

Explain where the program is located in the file "bootlbit.dump". In order to find the BRAM structure you need also "system_bd.bmm".

- In Impact Programer configure the FPGA with `download.bit` and verify that booting process from flash takes place.

- Create a PROM file (in .mcs format) ready for programming the platform flash memory or the BPI flash. Inspect the file in an ASCII editor (e.g. Jedit).
  PLEASE DO NOT REPROGRAM ANY FLASH MEMORY ON THE SPARTAN-3E BOARD!!!

Lab #01 b:

1.7 LAB Experiment: Vivado Embedded System Development and Boot Procedure

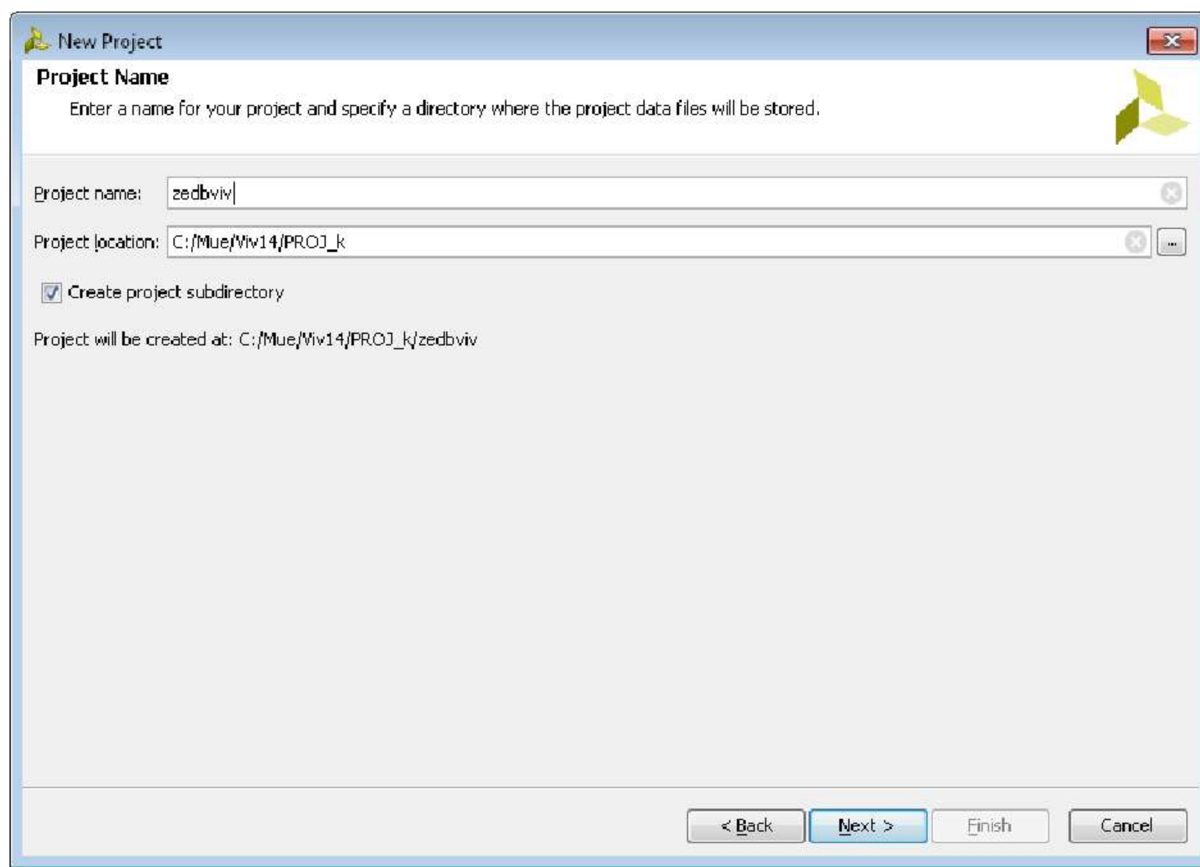
The ISE/EDK/SDK embedded development system will not fully support newer devices. Major changes to improve the efficiency could not be integrated in the old tools. The Vivado development tool offer better integration of simulator, the integrated logic analyzer and does not require EDK anymore. Perhaps its biggest improvement is the HLS option to develop algorithms in hardware with C, C++, or System-C.

Since Vivado is not an evolution of ISE/ISIM/ChipScope/EDK but a new development many things are completely different (i.e. constraints are now defined with industry standard XDC (Xilinx design constraints, almost the same to SDC – Synopsis design constraints).

This lab is a primer on Vivado system development (hardware and software) and shows the power-on boot procedure. Since for Zynq devices the processing system (CPU) boots first the booting process is much different from conventional FPGA systems.

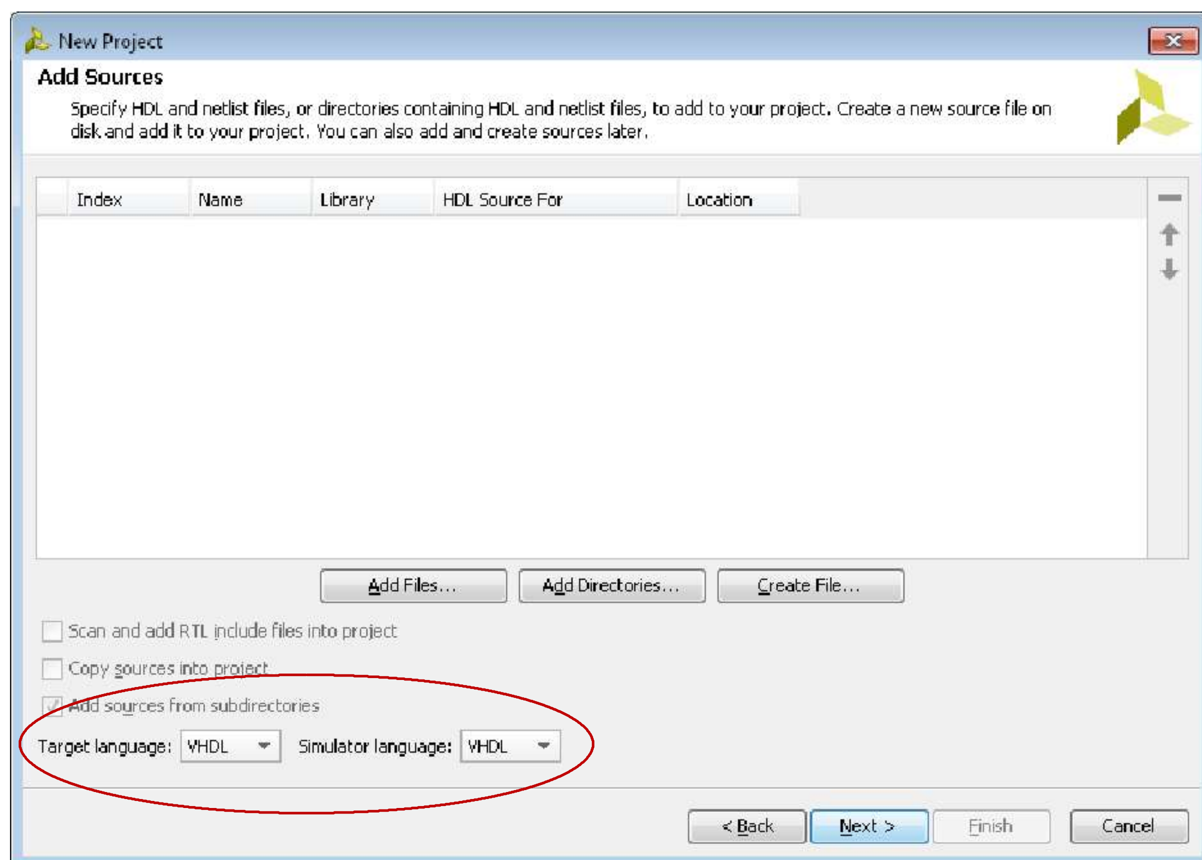
▶ Create a new project in the directory C:\mueller\ESPR\PROJ<k> where *k* is your group number, i.e. C:\mueller\ESPR\PROJ3.
The name of the project should be `zedbviv`.

(1) Create a new project with the `Create New Project` Wizard.



Press `Next` .
Select `RTL Project` as Project Type (should also be the default).
Do not specify sources at this time; press **Next** .

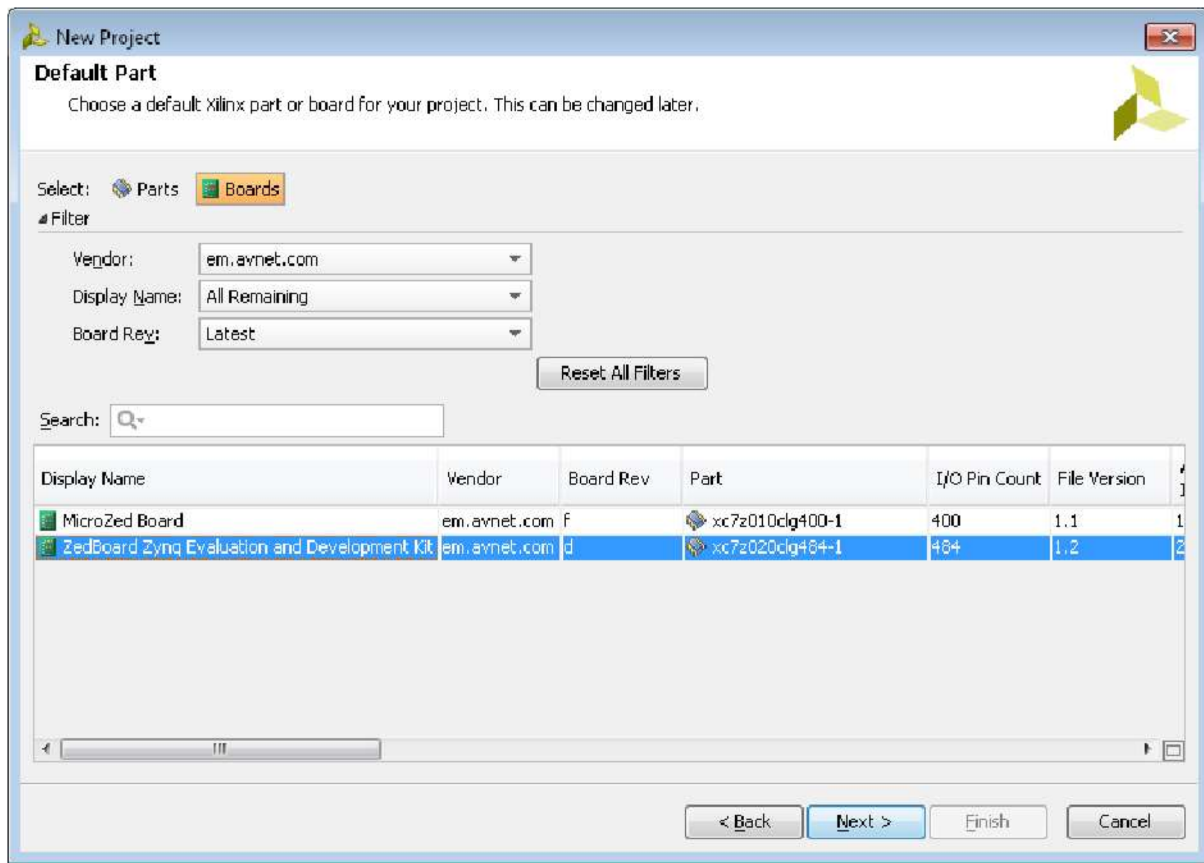
Select VHDL as target language and as simulator language; press **Next** .



Also press `Next` in the “Add Existing IP” dialog.

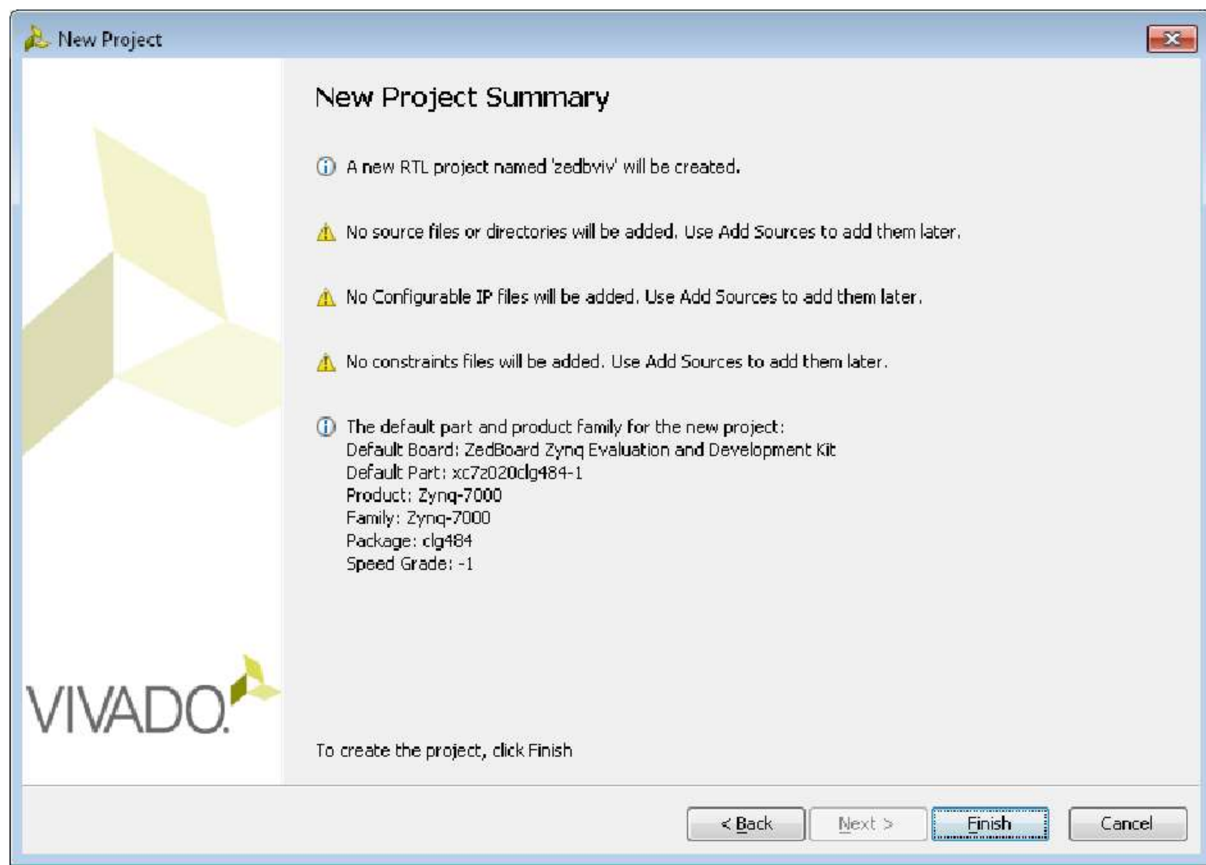
Press **Next** in the “Add Constraints” dialog.

- In the “Default Part” dialog select the ZedBoard from em.avnet.com and the `xc7z020clg484-1` FPGA.

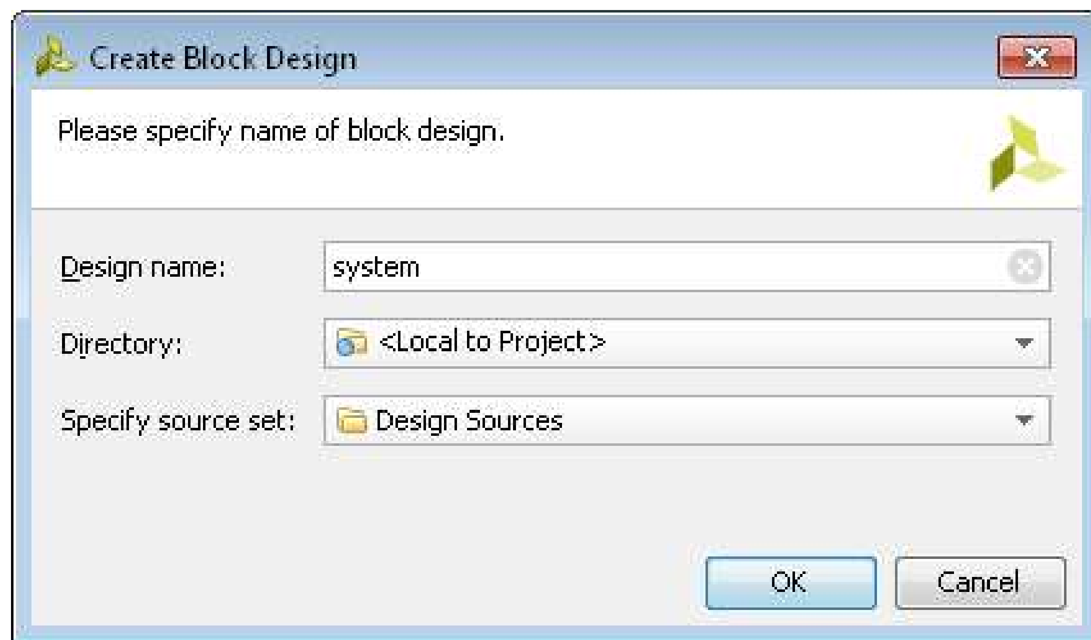


With this selection all board constraints will be available for design (i.e. interface to the DDR-3 memory, ethernet, and serial communication etc.).

▶ Press **Finish** in the “New Project Summary” page. The page should look like this:

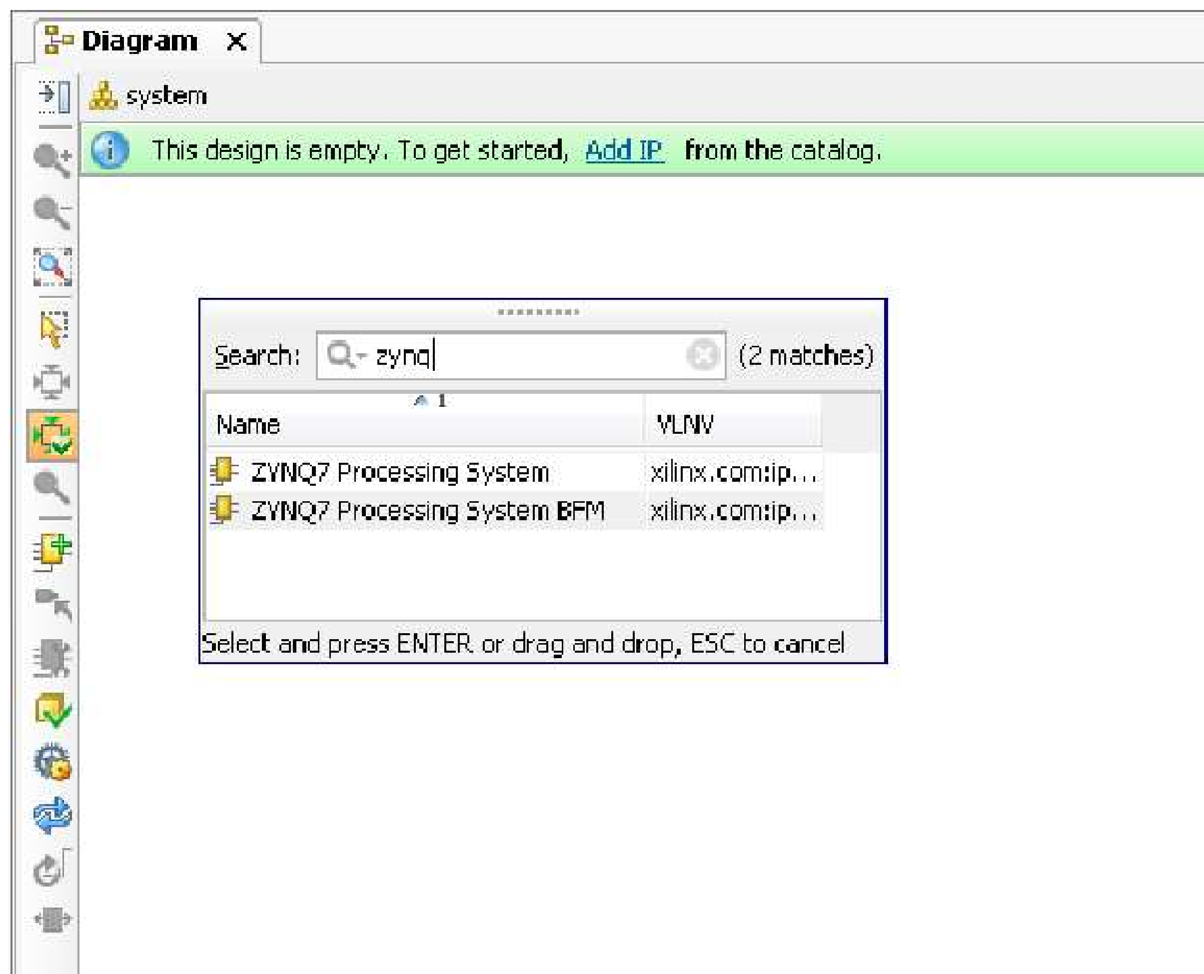


- Insert the ARM Cortex A9 processing system
  In the "Flow Navigator" pane click on
  `Create Block Design`
  set the Design name to `system`.

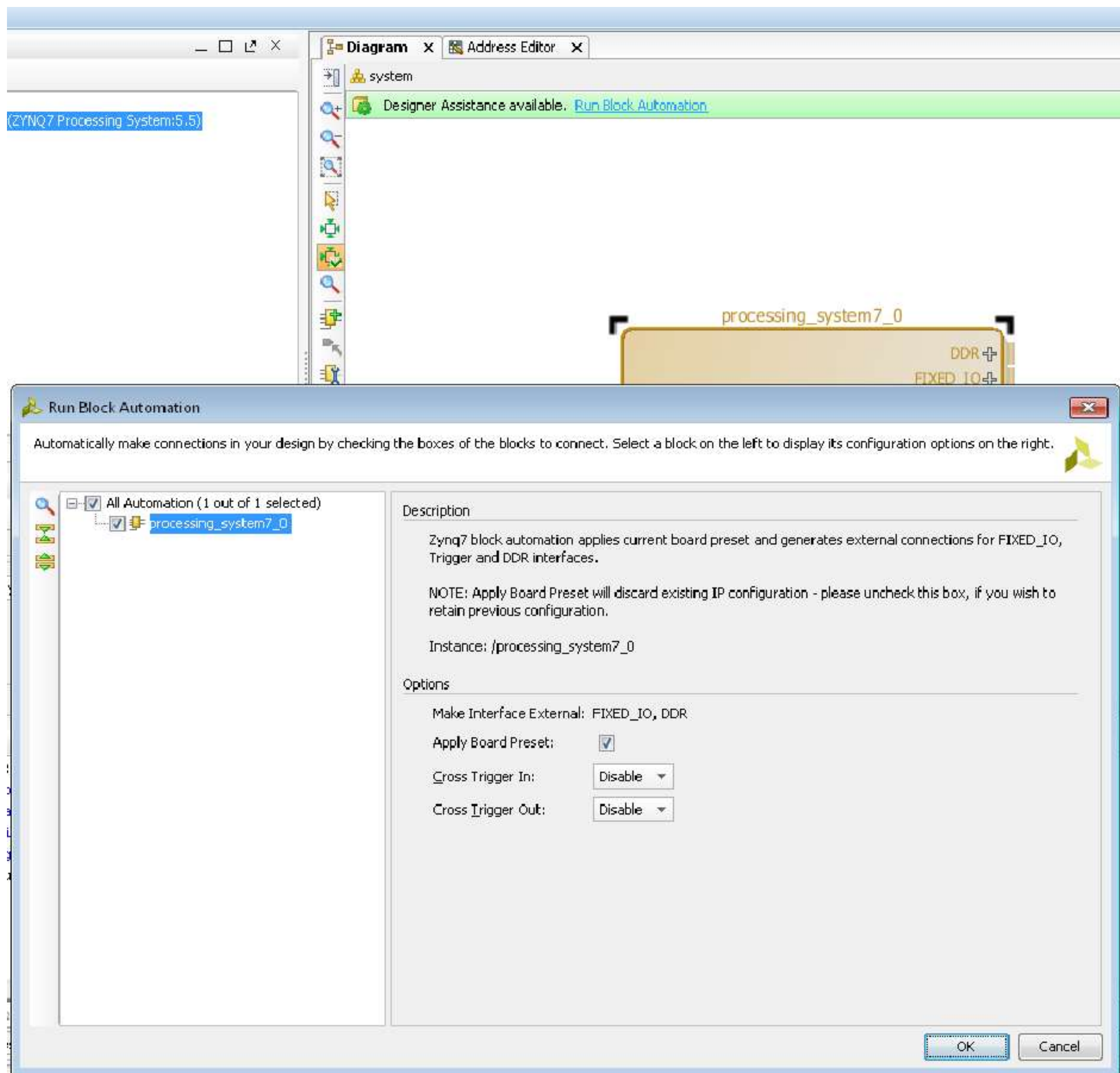


Press `OK` (leaving the other defaults).

▶ Select `Add IP` from the top row or from the left column.
Search for `znyq` ...
Select the `ZNYQ Processing System` (double-click)

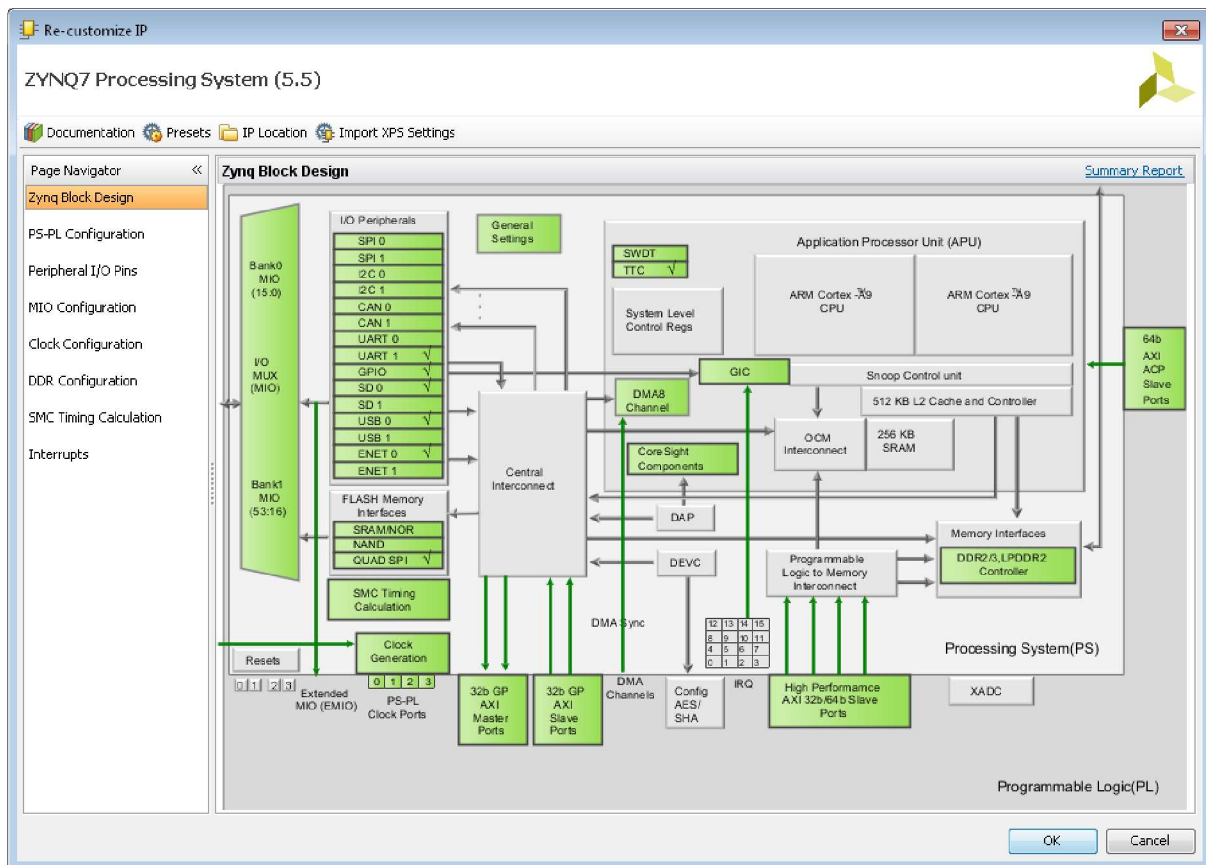


- Run Block Automation as suggested from the top row of the Block Design Diagram



Accept all defaults as shown and press OK .
Fixed_IO and DDR memory will be added. This is known from the board dialog page.

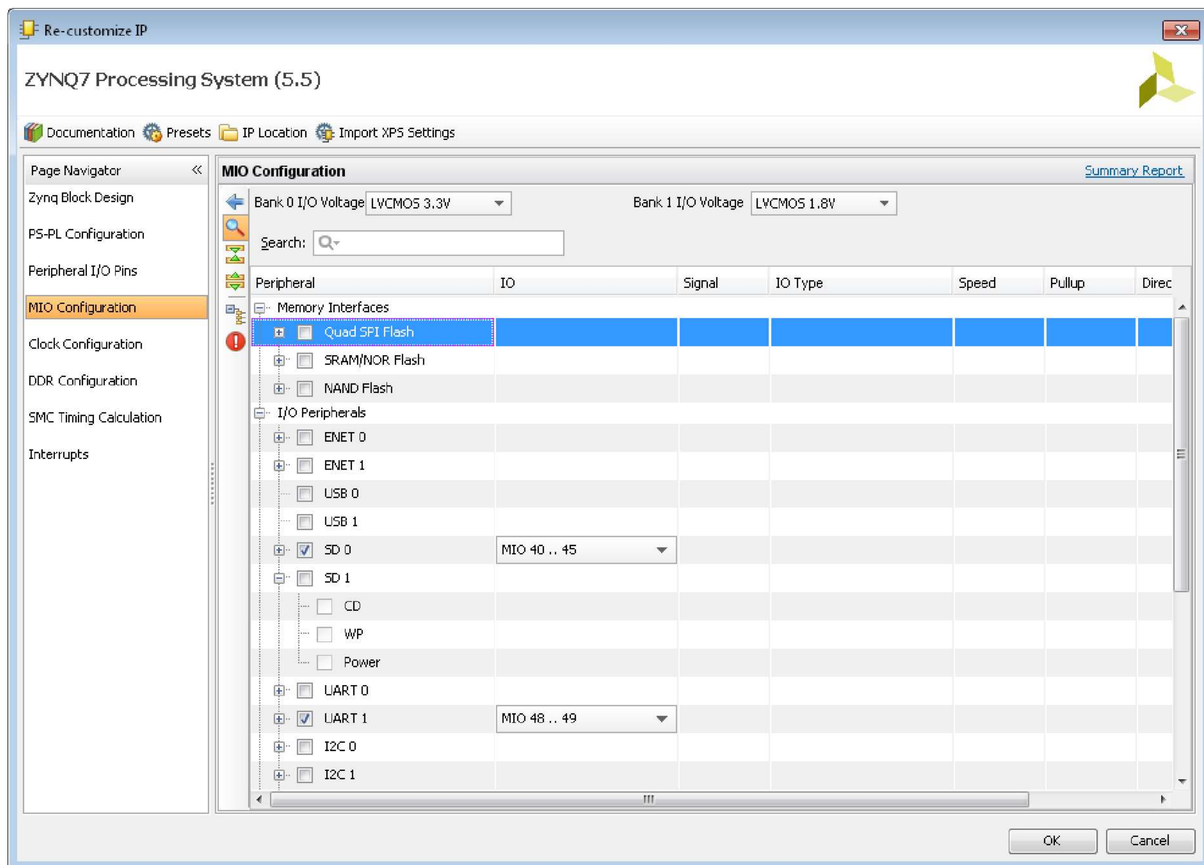
▶ Double-click on the ZYNQ block to open the customization window.



Select the `MIO Configuration` from the “Page Navigator” and deselect:

```
Quad SPI Flash
ENET 0
```

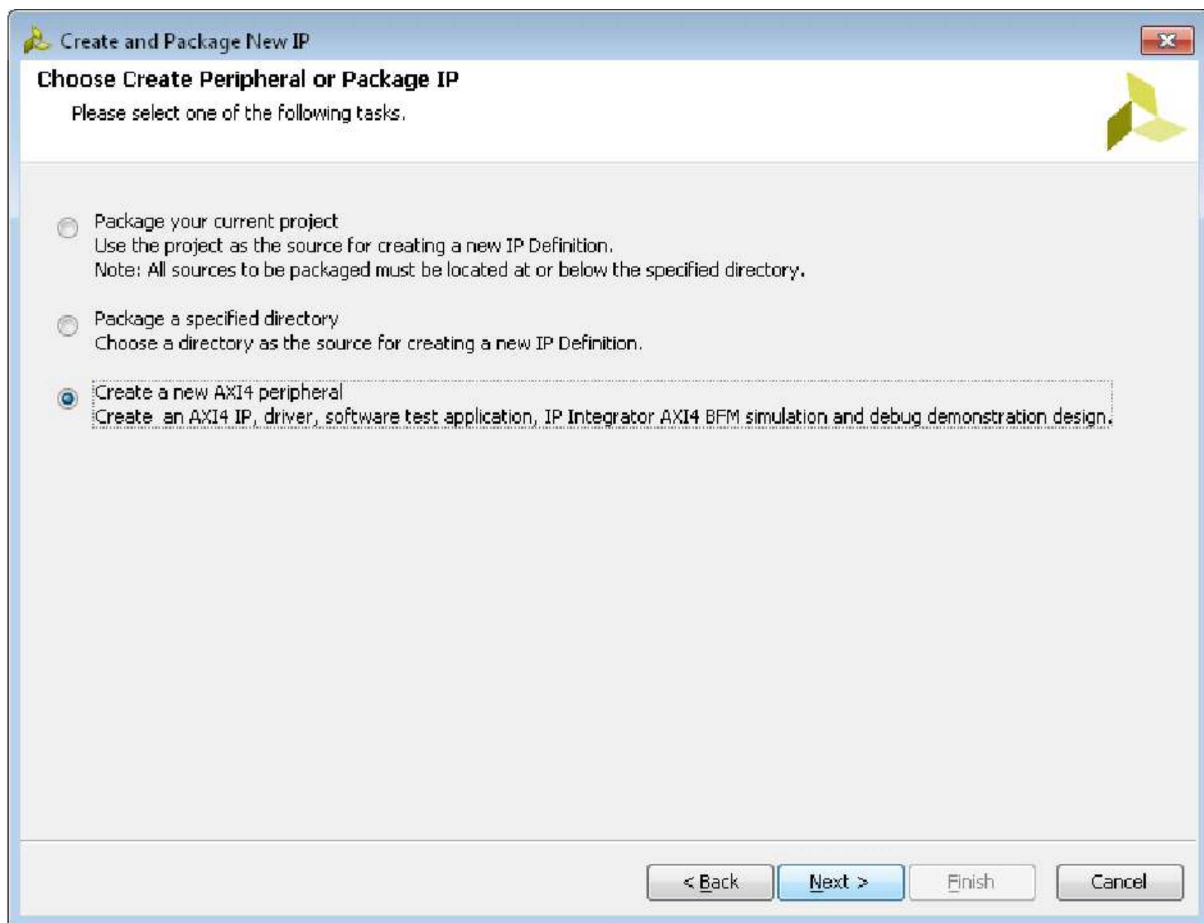
as we do not need them in our design.



It is important to keep UART 1 (for serial communication) and SD0 (for the SD card boot process).
Press **OK** to finish ZYNQ customization.

- Next we add custom IP to the ZYNQ processing system.
  From the main menu select `Tools > Create and Package IP`

  Click **`Next >`**.

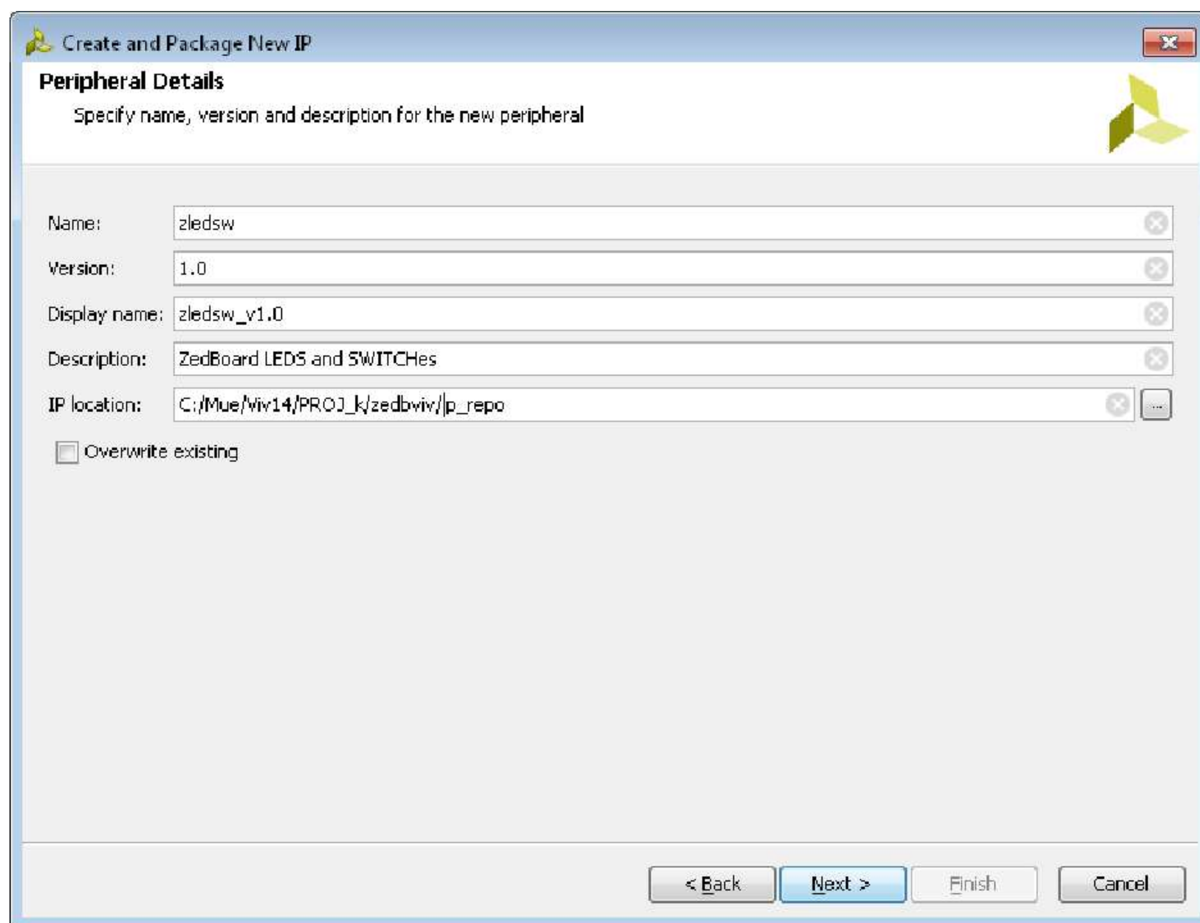


Select `Create A New AXI4 Peripheral` and click **Next >** .

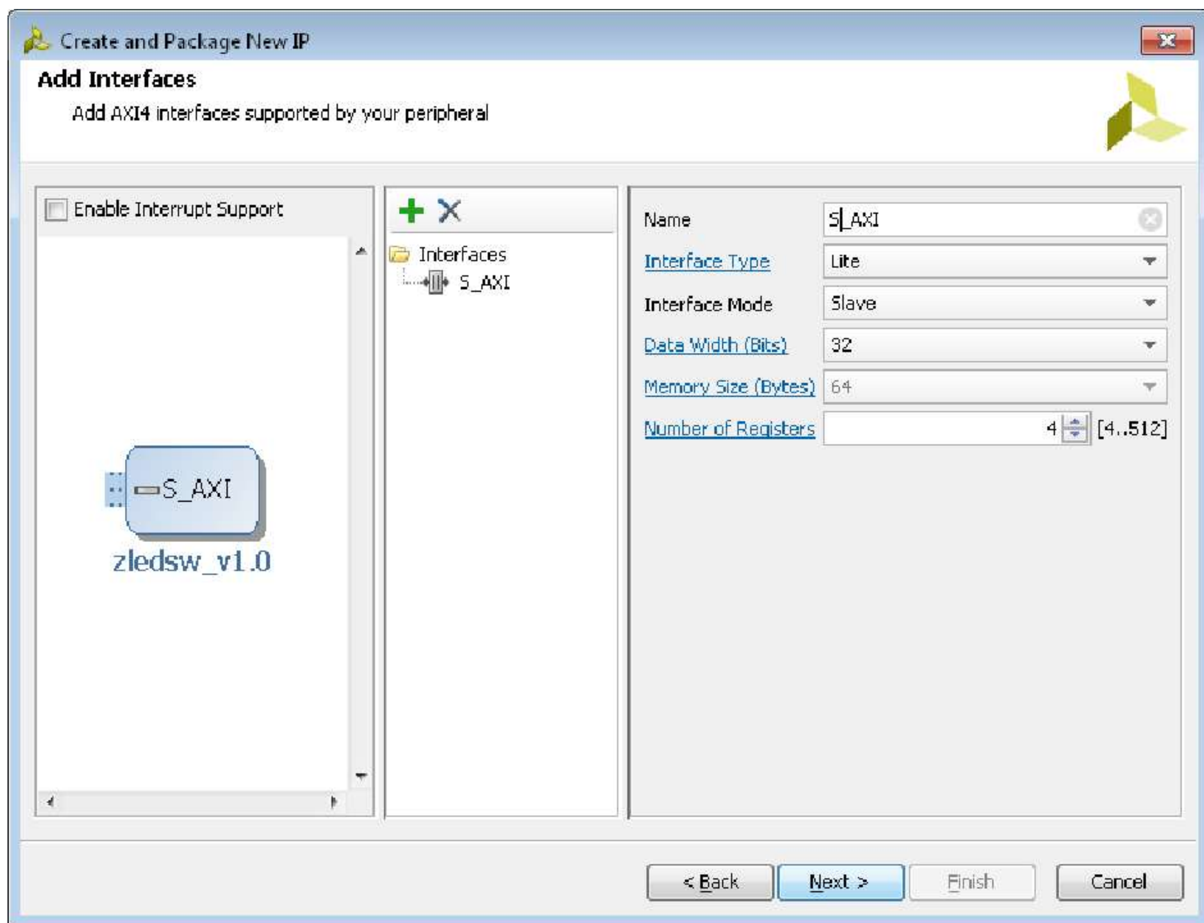
Change the IP Peripheral Details according to the following figure. The ip_repo folder should be in the project directory, i.e.

`C: /mueller/Viv14/PROJ_3/zedbviv/ip_repo` .

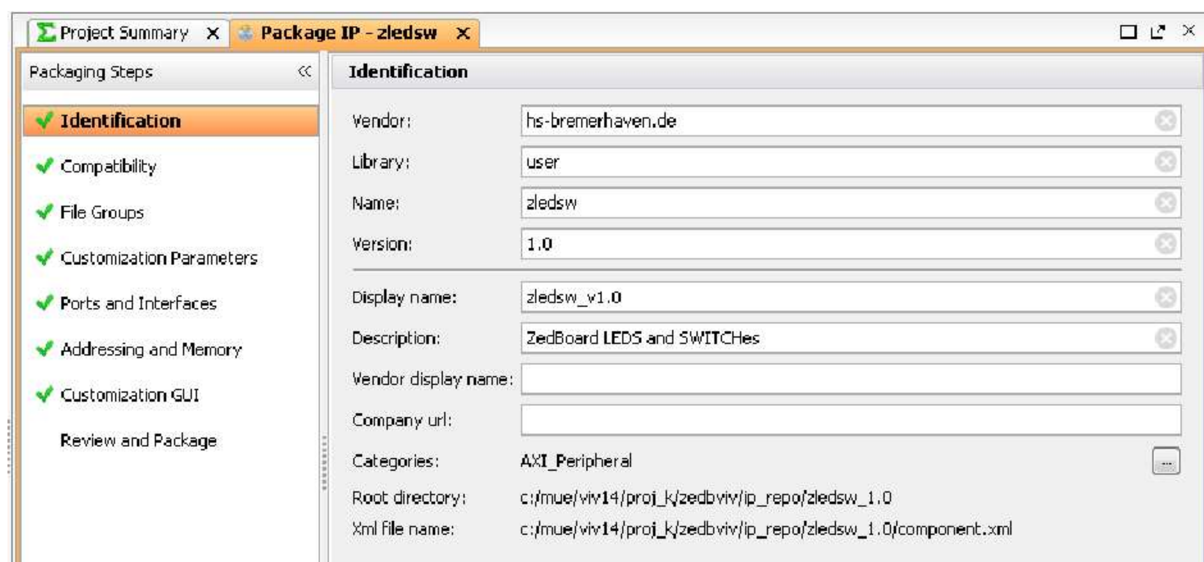
Click **Next >** .



Change the interface name to `S_AXI` and click **Next >** .



Select `Edit IP` and click **Finish** .



Change the Identification tags according to the above figure. Select `Edit IP` and click **Finish** .

In `File Groups` find the files `zledsw_v1_0_S_AXI.vhd` and `zledsw_v1_0.vhd`. These are the template file for the `zledsw` IP.
Open the top level file `zledsw_v1_0.vhd` and the implementation file `zledsw_v1_0_S_AXI.vhd` by double-clicking on their file names.

▶ Changes to `zledsw_v1_0.vhd`:
1. Add user generics (number of switches and leds)

```
-- Users to add parameters here
C_LED_WIDTH : integer := 8;
C_SWITCH_WIDTH : integer := 8;
-- User parameters ends
-- Do not modify the parameters beyond this line
```

2. Add user ports:

```
-- Users to add ports here
zLEDs : out std_logic_vector(C_LED_WIDTH-1 downto 0);
zSwitch : in std_logic_vector(C_SWITCH_WIDTH-1 downto 0);
-- User ports ends
-- Do not modify the ports beyond this line
```

3. In component declaration of `zledsw_v1_0_S_AXI` change the generics

```
component zledsw_v1_0_S_AXI is
generic (
U_LED_WIDTH : integer := 8;
U_SWITCH_WIDTH : integer := 8;
```

4. In component declaration of `zledsw_v1_0_S_AXI` add the ports

```
port (
LEDs : out std_logic_vector(U_LED_WIDTH-1 downto 0);
SWITCHes : in std_logic_vector(U_SWITCH_WIDTH-1 downto 0);
```

5. In the intstantiation of `zledsw_v1_0_S_AXI` add generic maps

```
U_LED_WIDTH => C_LED_WIDTH,
U_SWITCH_WIDTH => C_SWITCH_WIDTH,
```

6. In the instantiation of `zledsw_v1_0_S_AXI` add port maps

```
LEDs        => zLEDs,
SWITCHes   => zSwitch,
```

▶ Changes to `zledsw_v1_0_S_AXI.vhd`:
1. Add genericsto the entity `zledsw_v1_0_S_AXI`

```
-- Users to add parameters here
U_LED_WIDTH : integer := 8;
U_SWITCH_WIDTH : integer := 8;
-- User parameters ends
-- Do not modify the parameters beyond this line
```

2. Add user ports:

```
-- Users to add ports here
LEDs : out std_logic_vector(U_LED_WIDTH-1 downto 0);
SWITCHes : in std_logic_vector(U_SWITCH_WIDTH-1 downto 0);
-- User ports ends
-- Do not modify the ports beyond this line
```

3. Change the beginning of the read process in order to read the switches:

```
process (slv_reg0, slv_reg1, slv_reg2, slv_reg3, axi_araddr,
         S_AXI_ARESETN, slv_reg_rden,
         SWITCHes)
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
    -- Address decoding for reading registers
    loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto
                                                  ADDR_LSB);
    case loc_addr is
      when b"00" =>
        reg_data_out <= slv_reg0(31 downto 8) & SWITCHes;
```

3. At the end of the file add (for writing to LEDs):

```
-- Add user logic here
LEDs <= slv_reg0(7 downto 0);
-- User logic ends
```

▶ Click on "Run Synthesis" in the Flow Navigator **of the edit_zledsd project:**
If you encountrer errors, correct them!
If there are no errors click Cancel after the synthesis run.

▶ Package the IP

1. Select the package IP page and navigate to "Customization Parameter"
   Click on the "merge Customization parameters wizard".

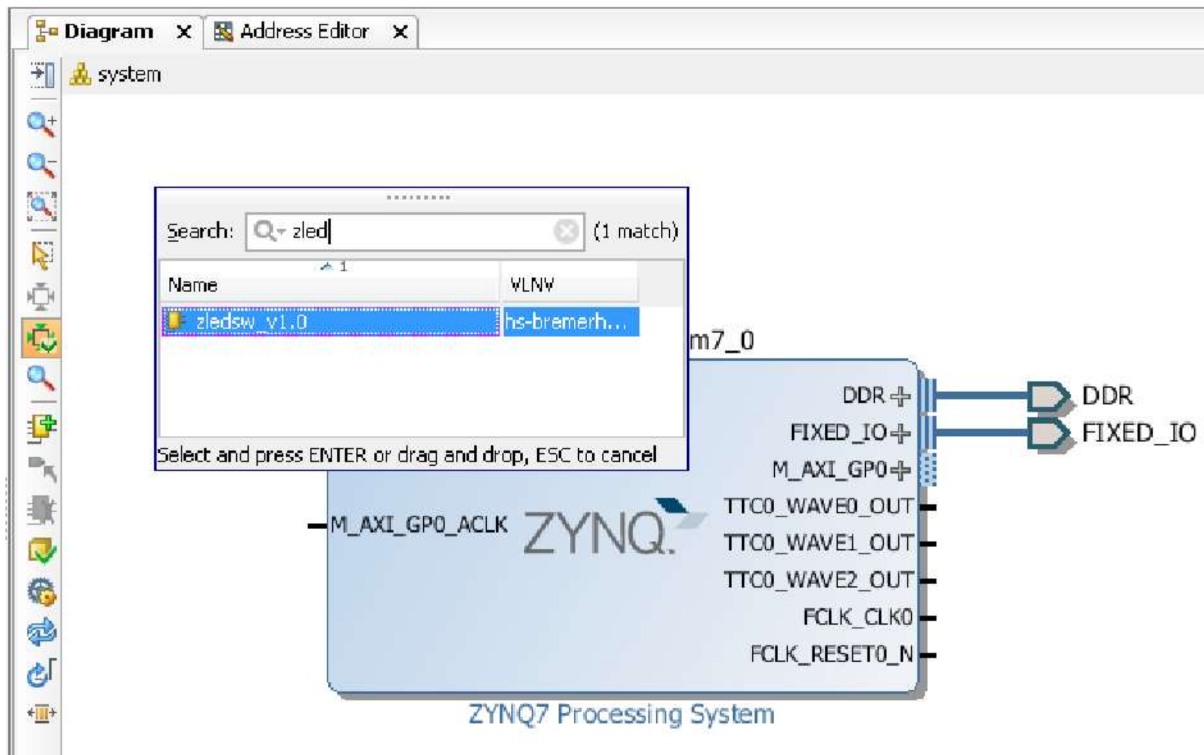
2. Verify that `zLEDs` and `zSWITCHes` are part of "Ports and Interfaces".

3. Verify that `zLEDs` and `zSWITCHes` are part of the "Customization GUI".
   Execute the "merge Customization GUI wizard" if this option is offered.

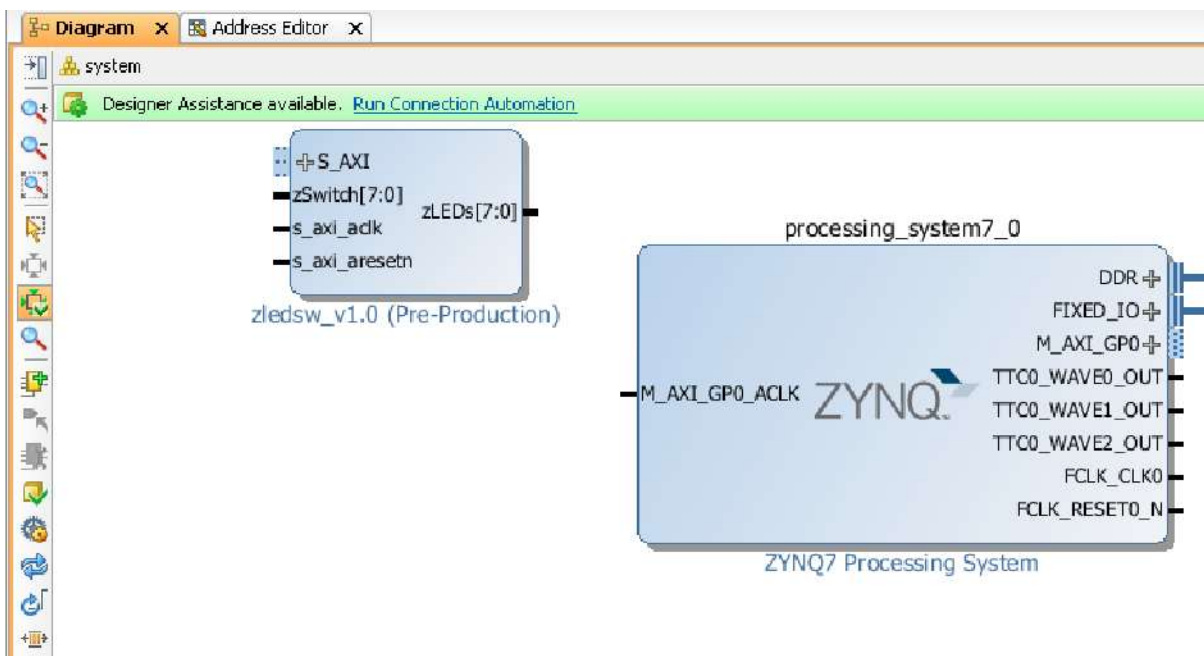
4. Under "Review and Package" click on **Re-Package IP** . This will create the
   repository (user) to use this IP in the design. This usually closes the project.

▶ In project `zebbviv` return to the `Block Design Diagram`

1. Click on **Add IP**
   Search for `zled`



2. Double-click on the zledsw_v1.0 instance which adds the IP to your design.



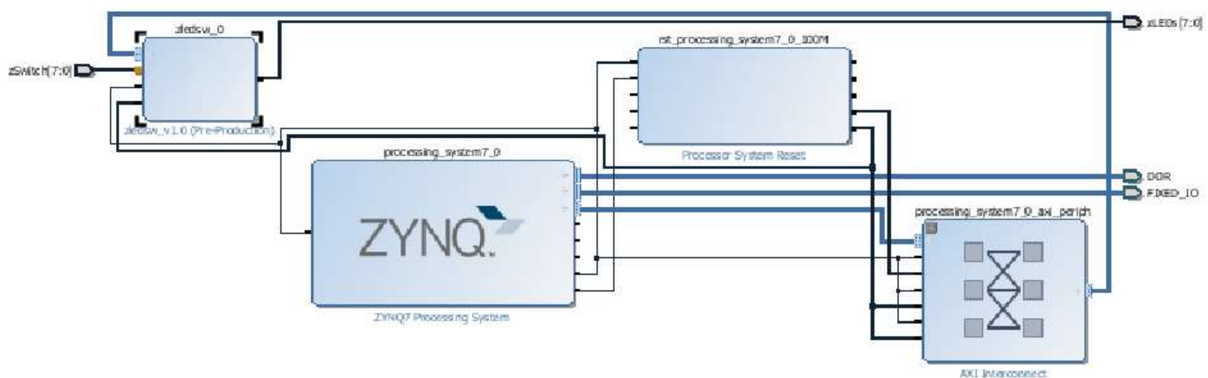
3. “Run Connection Automation”

`AXI Interconnect` and the `Processor System Reset` will be added automatically.

4. Select the `zLEDs` port and with a right-click make it external.

5. Select the `zSWITCHes` port and with a right-click make it external.

Finally the diagram should look like this:

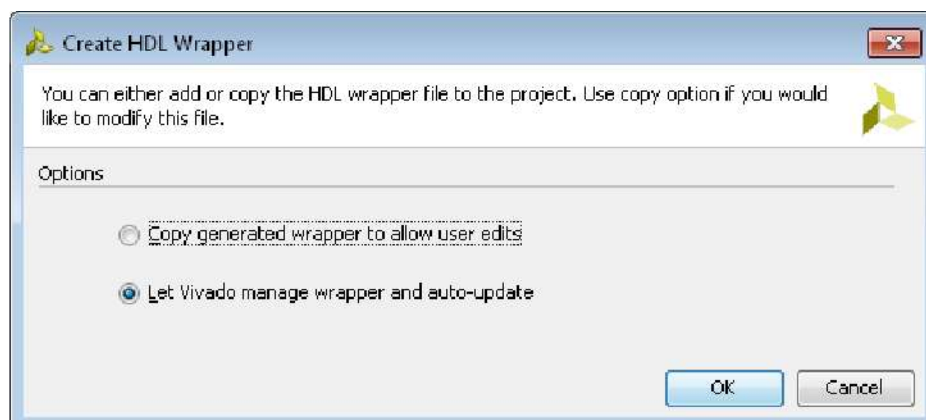


6. Check on the "Address Editor" tab that `zledsw_0` is accessible.

▶ Select **Tools** > **Validate Design** from the main menu.
Be sure that no errors are reported.

▶ In the "Sources" view right click on system.bd and select **Create HDL Wrapper**.

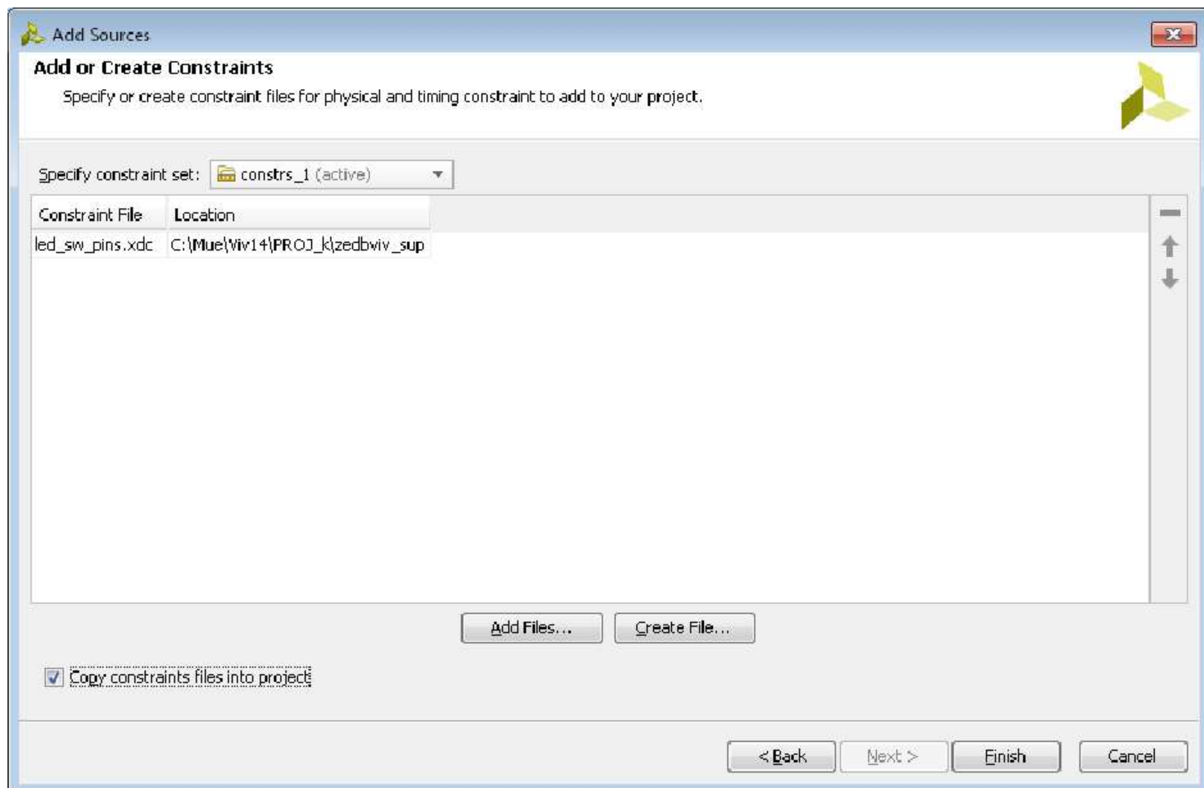
Accept the following defaults and click **OK**.



▶ In the "Sources" window right-click on `constrs_1` and select **Add Sources**.

Select "Add or Create Constraints".

Navigate to the support folder and select `led_sw_pins.xdc`.
Be sure to check the "Copy Constraints Files into Project" option (see below).



Below is the top of the `led_sw_pins.xdc` file:

```
# ZedBoard xdc
# zLEDs [7:0]
###########################
# On-board led            #
###########################
set_property PACKAGE_PIN T22 [get_ports zLEDs[0]]
set_property IOSTANDARD LVCMOS33 [get_ports zLEDs[0]]
set_property PACKAGE_PIN T21 [get_ports zLEDs[1]]
set_property IOSTANDARD LVCMOS33 [get_ports zLEDs[1]]
set_property PACKAGE_PIN U22 [get_ports zLEDs[2]]
set_property IOSTANDARD LVCMOS33 [get_ports zLEDs[2]]
set_property PACKAGE_PIN U21 [get_ports zLEDs[3]]
set_property IOSTANDARD LVCMOS33 [get_ports zLEDs[3]]
```

▶ In the "Flow Navigator" click on **Run Synthesis**.

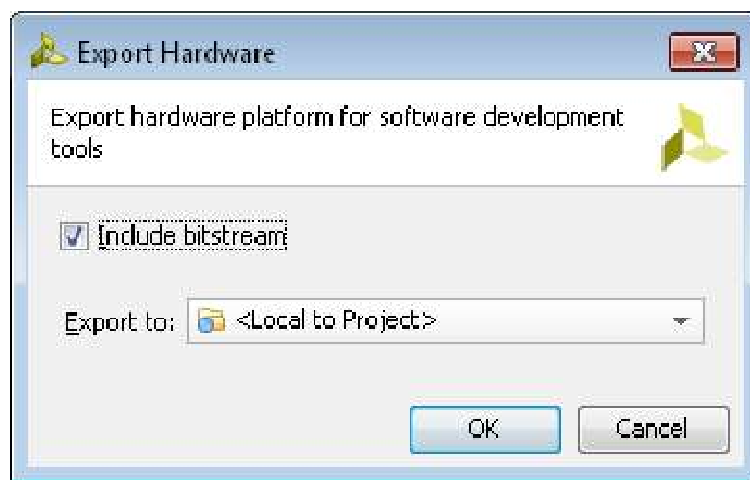
Select “open Synthesized Design” on completion.
Verify that the external pins are routed correctly.

▶ In the “Flow Navigator” click on **Generate Bitstream** .

This involves Implementation phase. If you are asked accept the Implementation run. You may check the Implemented Design in the `Device` tab.

▶ IMPORTANT: Open the block design (”Open Block Design” in the Flow Navigator)

Select **File > Export > Export Hardware** while the Diagram is open.
Make sure that **“Include Bitstream”** ***has been marked.***



▶ Select **File > Launch SDK**
and accept the defaults.
You may now close Vivado and operate with SDK.

▶ Create a Board Support Package
Select **File > New > Board Support Package** and provide the name `zbiptst_bsp_0`. Don’t add any support libraries since they are not needed.

▶ Create a new Application `zbiptest` and select the Helloworld template.
Use the Board Support Package `zbiptst_bsp_0` from the previous step.

Rename `helloworld.c` to `zbipmain.c` .

▶ Change the code to

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
```

```
int main()
{
    init_platform();
    print("--- zbiptest V0.0a ---\n");
    print("Thank you for using zbiptest.\n");
    cleanup_platform();
    return 0;
}
```

Verify that it executes properly on the ZYNQ7020.
Open a PuTTY terminal with baud rate 115200 and the correct com port (can be seen from the windows device manager). The output should be as follows:

```
--- zbiptest V0.0a ---
Thank you for using zbiptest.
```

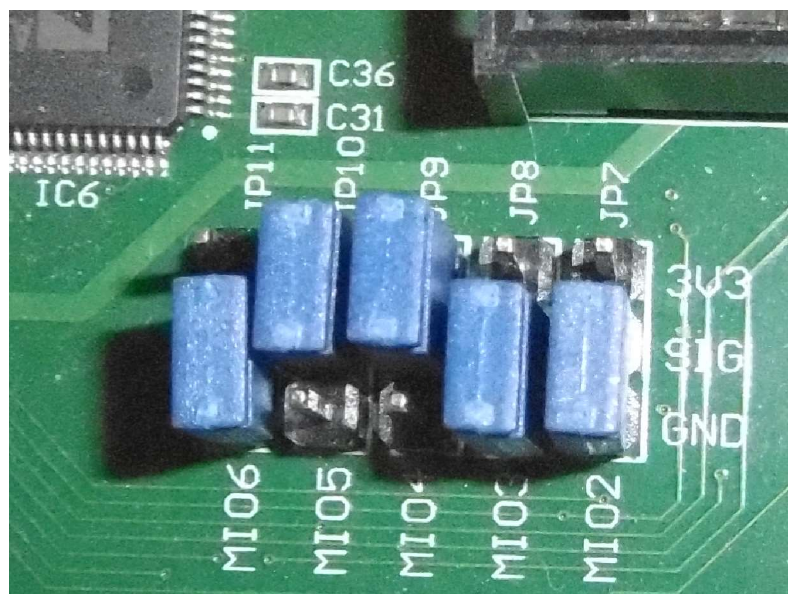
Add the following code to verify access to LEDs and SWITCHes.

```
#include <stdio.h>
#include "xil_printf.h"
#include "platform.h"
#include "xil_printf.h"
#include "xil_types.h"
#include "xparameters.h"


#define LEDSW_REG(k)  (*(volatile u32 *)(XPAR_ZLEDSW_0_S_AXI_BASEADDR+4* k))

int main()
{
    u32  x;

    init_platform();
    print("--- zbiptest V0.0a ---\n");
    x = LEDSW_REG(0);
    xil_printf("Switches = %x\n\r", x);
    LEDSW_REG(0) = x;
    print("Thank you for using zbiptest.\n");
    cleanup_platform();
    return 0;
}
```

This application should be executed on power-on reset without SDK.

Booting the ZYNQ device involves several stages:

1. The Zero Stage Bootloader executes.
   This bootloader is stored in ROM of the ARM core and cannot be changed.



If the MIO connected jumpers are set as above the first stage bootloader searches for
boot.bin
on a FAT32 formatted SD card. This is a single file but it is in fact a container file with must contain the
First Stage Bootloader (FSBL) and many other data files.

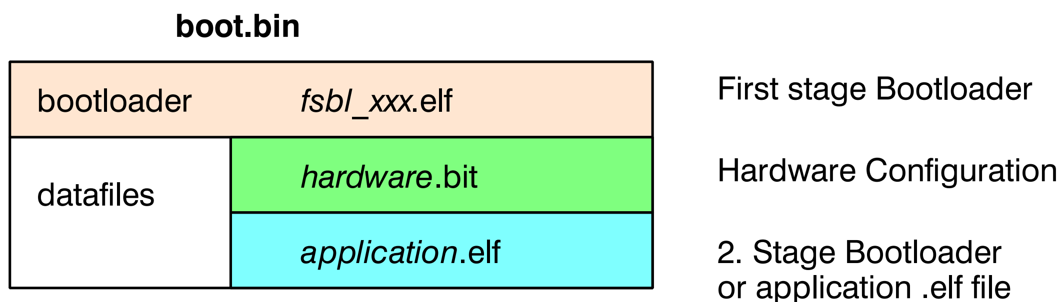


Figure 1.10: boot.bin file structure

The boot file name must be boot.bin while the filenames inside boot.bin are arbitrary.

- Create a FSBL with the “Application Wizard”
  1. Select **File > New > Application Project**
  2. Choose name `zbfsbl_0`
  3. Allow the wizard to create its own Board Support Package. This is required to access the FAT32 SD card file structure.
  4. Select `Znyq FSBL` on the Templates page
  5. Click **Finish** . This takes a long time since the FSBL is a complex software.

- Open the file `fsbl_debug.h` from the source folder of `zbfsl_0` .

```
#ifdef __cplusplus
extern "C" {
```

```
#endif

#define FSBL_DEBUG_INFO     // <=== this line

#define DEBUG_GENERAL        0x00000001    /* general debug  messages */
#define DEBUG_INFO  0x00000002    /* More debug information */

#if defined (FSBL_DEBUG_INFO)
#define fsbl_dbg_current_types ((DEBUG_INFO) | (DEBUG_GENERAL))
#elif defined (FSBL_DEBUG)
#define fsbl_dbg_current_types (DEBUG_GENERAL)
#else
#define fsbl_dbg_current_types 0
#endif
```

Do a **Clean Project** on zbfsbl_0 to rebuild `zbfsl_0.elf` .

Create a .bif script and build `boot.bin` .
Create a new folder `flash` under the .sdk folder, i.e.

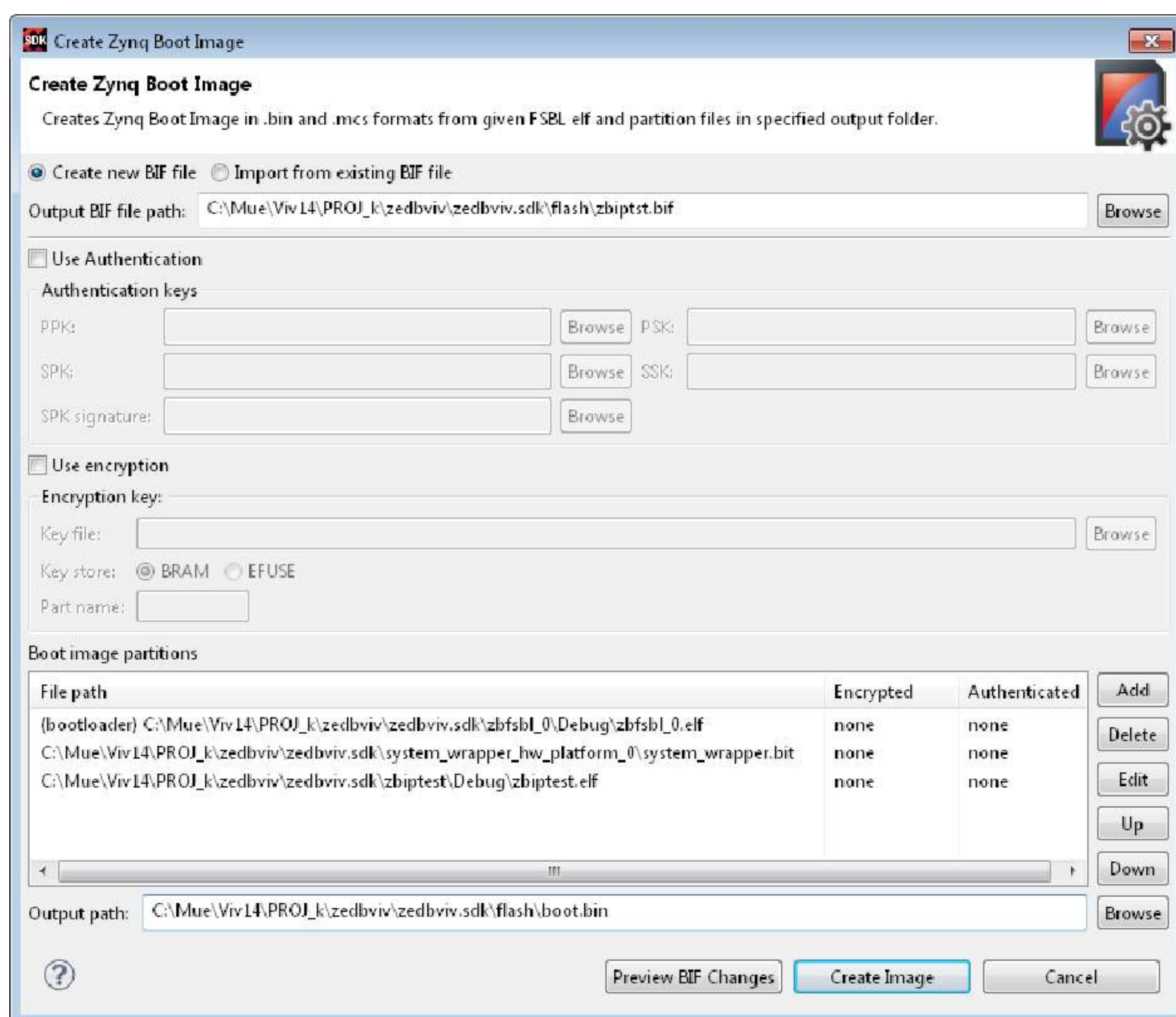
`C: \mueller\ESPR\PROJ3\zedbviv\zedbviv.sdk\flash`

Select **Xilinx Tools** > **Create Zynq Boot Image** .
Set up the Boot Image page as follows:

Click on **Create Image** to start bootgen.

Verify that the `zbiptst.bif` file conatins the following text (or equivalent file names if your file names differ).

```
the_ROM_image:
{
    [bootloader]C:\Mue\Viv14\PROJ_k\zedbviv\zedbviv.sdk\zbfsbl_0\Debug\...
                      zbfsbl_0.elf
    C:\Mue\Viv14\PROJ_k\zedbviv\zedbviv.sdk\...
        system_wrapper_hw_platform_0\system_wrapper.bit
    C:\Mue\Viv14\PROJ_k\zedbviv\zedbviv.sdk\zbiptest\Debug\zbiptest.elf
}
```



- Copy `boot.bin` to the root file system of a FAT32 formatted SD card.
- Power off the ZedBoard and insert the SD card.
  Prepare the start of PuTTY (you cannot start it right away because the serial device does not exist on the PC if power is down).
- Power in ZedBoard and start PuTTY. Watch out for the following messages:

```
Xilinx First Stage Boot Loader
Release 2014.3  Oct 12 2014-21:44:22
Devcfg driver initialized
Silicon Version 1.0
Boot mode is SD
SD: rc= 0
SD Init Done
Flash Base Address: 0xE0100000
Reboot status register: 0x60000000
Image Start Address: 0x00000000
Partition Header Offset:0x00000C80
```

```
Partition Count: 3
Partition Number: 1
...
...
...
SUCCESSFUL_HANDOFF
FSBL Status = 0x1

--- zbiptest V0.0a ---
Switches = 52
Thank you for using zbiptest.
```

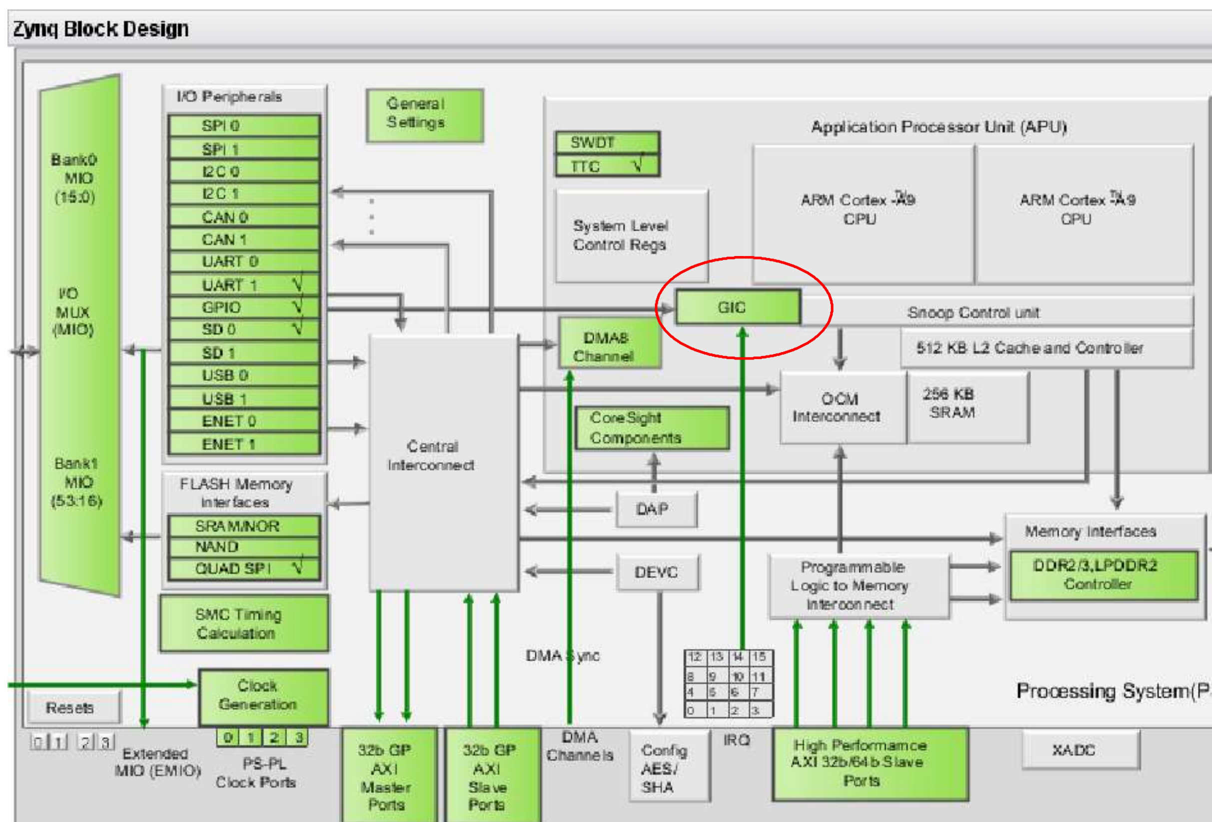
***

Lab #01c:

1.8 LAB Experiment: Zynq Interrupt

Interrupts are essential for real-time software. Interrupt-driven software is the fastest way to respond to events. All operating systems kernels are using interrupts to schedule processes or tasks. In bare metal applications interrupts allow to respond to asynchronous events or to provide precise timing for control systems of discrete filter computation.

Interrupt controller hardware is required to pass interrupts to the CPU and take care of priorities and enable/disable functionality. In pure FPGA systems an interrupt IP block needs to be added to the system. Zynq devices have an integrated controller called GIC (**g**eneric **i**nterrupt **c**ontroller). For external interrupts (i.e. AXI interrupts) the GIC needs to be configured from the property Zynq property sheet below. If only internal interrupt sources (GPIO or timers/watchdog) the configuration is possible from software (SDK).



▶ In this lab a timer interrupt from the internal timer (scutimer) should be set up required for obtaining precise sampling time in control systems and FIR filter

computations. This lab uses a timer local to one core of the ARM A9 dual core CPU.

The interrupts should be added to the previous lab with the custom IP block which controls LEDs and SWITCHes.

Fully functional timer interrupts involves the following steps:

- create an interrupt controller instance
- search for the interrupt device base on its device id
- create and initialize the interrupt controller instance
- create the hardware controller handler (exception handler)
- search for the time device base on its device id
- create and initialize the time instance
- program timer mode (down counter with auto reload the timer value)
- load timer (initial and reload value)
- write an interrupt handler with standard exception handler structure
- install the (timer) interrupt handler
- enable the timer interrupt source in the GIC
- enable the interrupt in the timer
- enable processor interrupts
- start the timer
- a shutdown sequence should be provided when the program shuts down (makes several restarts safe):
    - o stop timer
    - o disable exceptions
    - o disable interrupts in the GIC

▶ Create a new application called `zbisrtst` (in SDK!).
You may start with the a “Helloworld” application. Rename `helloworld.c` to `zbisrmain.c` .

▶ Add header files to your application (if not already included):

```
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "xscutimer.h"
```

▶ define macro to access user logic:

```
#define LEDSW_REG(k) (*(volatile u32 *)(XPAR_LEDSW_0_S_AXI_BASEADDR + 4 * k))
```

You may need to adjust the IP name if you selected a different name.

Device (IP) initialization usually involves two steps:

(1) Call of a lookup function by a unique device name from the device table. This table is part of the hardware description from Vivado or EDK.
(2) Call to a device initialization function which set a device object (a structure in “C”).

For both operations variables have to be provided:

```
(1) static <Device>_Config  *devconfig_pointer;
(2) static <Device>_name   device_obj;
```

(1) is only a pointer to device data while (2) is a structure which hold all device configuration information. A pointer to this structure will be passed to all library functions for this particular device.

The initialization calls are:

```
(1) devconfig_pointer = <Device>_LookupConfig(DEVICE_NAME);
(2) <Device>_CfgInitialize(&device_obj,
      devconfig_pointer, devconfig_pointer->BaseAddr);
```

It is recommended to follow the same initialization strategy for your own IPs.

- Create device pointer and objects for the interrupt controller

```
// ---- interrupt controller -----
static XScuGic  Intc;                  // interrupt controller instance
static XScuGic_Config  *IntcConfig;    // configuration instance
```

- Create device pointer and objects for the scutimer

```
// ---- scu timer -----
static XScuTimer  pTimer;                // private Timer instance
static XScuTimer_Config  *pTimerConfig;  // configuration instance
#define TIMER_LOAD_VALUE  66666667       // what time is this?
```

- Define the timer count value. The timers clock is half of the ARM cpu clock (666.666 MHz), i.e. timer clock is 333.333 MHz.

```
#define TIMER_LOAD_VALUE  nnnnnnnnn    // 200ms
```

Replace `nnnnnnnnn` with the number that represents 200 ms.

- Create variables for communication with the interrupt service routine:

```
static volatile int  ISR_Toggle = 0, ISR_Count = 0;
```

This verifies that the interrupt service routine executes properly.

- In the main() function add the call

```
Xil_ExceptionInit();
```

to initialize the hardware interrupt driver. This function has no effect on ZYNQ but may be required for MicroBlaze or PowerPC.

All functions are called without evaluating the return status. For debugging purposes the following code can be used:

```
Status = xxx_CfgInitialize(...);
if ( Status != XST_SUCCESS) {
     print("function XXX failed\n");
}
```

- Initialize GIC (interrupt controller)

```
IntcConfig = XScuGic_LookupConfig(XPAR_SCUGIC_0_DEVICE_ID);
XScuGic_CfgInitialize(&Intc, IntcConfig, IntcConfig->CpuBaseAddress);
```

- Connect the interrupt controller hardware driver

```
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT,
          (Xil_ExceptionHandler)XScuGic_InterruptHandler, &Intc);
```

- Initialize and configure timer

```
pTimerConfig = XScuTimer_LookupConfig(XPAR_XSCUTIMER_0_DEVICE_ID);
XScuTimer_CfgInitialize(&pTimer, pTimerConfig, pTimerConfig->BaseAddr);
XScuTimer_EnableAutoReload(&pTimer);
XScuTimer_LoadTimer(&pTimer, TIMER_LOAD_VALUE);
```

- Write an interrupt service routine (isr). A possible template for this function is:

```
/*
 * ------------------------------------------------------------
 * Interrupt handler (ZYNQ private timer)
 * ------------------------------------------------------------
 */
static void TimerIntrHandler(void *CallBackRef)
{
    XScuTimer *TimerInstance = (XScuTimer *)CallBackRef;

    XScuTimer_ClearInterruptStatus(TimerInstance);
    ISR_Count++;
}
```

Add some change of LEDs to this function to verify that it is executed.

- In the `main()` function connect the isr with the timer interrupt

```
XScuGic_Connect(&Intc, XPAR_SCUTIMER_INTR,
          (Xil_ExceptionHandler)TimerIntrHandler, (void *)&pTimer);
```

- Enable timer interrupts in the GIC:

```
XScuGic_Enable(&Intc, XPAR_SCUTIMER_INTR);
```

- Enable the interrupts in the timer

```
XScuTimer_EnableInterrupt(&pTimer);
```

- Enable the interrupt handling in the ARM cpu:

```
Xil_ExceptionEnable();
```

- Start timer count (the interrupt service routine should execute now!):

```
XScuTimer_Start(&pTimer);
```

- Write a loop (while interrupts are processed) and create a termination condition:

```
    do {
          print("isrtst >>> ");  fflush(stdout);
          ...
          ...
          ...
    } while (keep_running != 0);
```

- After the loop do proper shutdown (reverse order of starting):

```
    XScuTimer_Stop(&pTimer);
    Xil_ExceptionDisable();
    XScuTimer_DisableInterrupt(&pTimer);
    XScuGic_Disable(&Intc, XPAR_SCUTIMER_INTR);
    print("Thank you for using  ISR Test.\n\r");
    cleanup_platform();
    return 0;
```

Disabling all devices allow a clean restart of the program after applying changes to the source code.

- Execution of `zbisrtst` should result in an output similar to the following text and the LEDs should light on and off:

```
-- ISR Test V0.0a ---
 * initialize exceptions...
 * lookup config GIC...
 * initialize GIC...
 * connect interrupt controller handler...
 * lookup config scu timer...
 * initialize scu timer...
 * Enable Auto reload mode...
 * load scu timer...
 * set up timer interrupt...
 * enable interrupt for timer at GIC...
 * enable interrupt on timer...
 * enable processor interrupts...
Switches = 51
 * start timer...
isrtst >>>
command was []
```

```
0  A
1  0
2  0
3  0
4  2
 => ISR_Count = 18
isrtst >>> x
Thank  you for using  ISR Test.
```

***

Interface Hardware

2 Pulse Oximeter Interface Hardware

The Pulse Oximeter is a medical device for measuring the oxygen saturation (S_pO_2) in blood. The measurement is based on the absorption of read an infrared light which depends in the amount of oxygen bound on the red blood cells. Moreover, it allows to plot an ECG-like diagram of the heat activity

While DSP is done in the FPGA, an interface for the optical finger probe is required. The interface requires a full PMOD connector (12 pins / 8 signals).

2.1 Finger Probe

The clip contains a red (660 nm) led and an infrared (895 nm) led. It also contains a pin diode to detect the intensity of red and infrared light (one pin diode for both wave lengths).

Figure 1.11: Pulse Oximeter finger probe

2.2 Interface Block Diagram

the following figure show the block diagram of the interface; the components are presented in detail later.

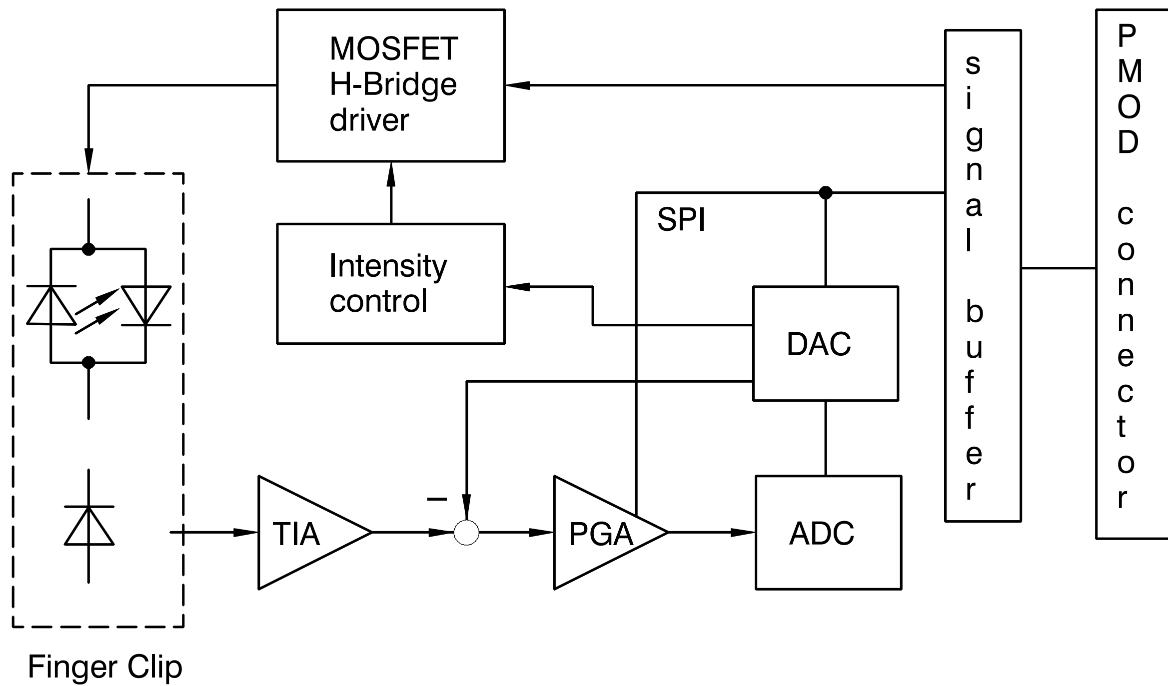


Figure 1.12: Pulse Oximeter interface block diagram

The elements from the block diagram 1.12 are independent. The interface has a simple design since all signal processing is carried out in the FPGA or in the microcontroller.

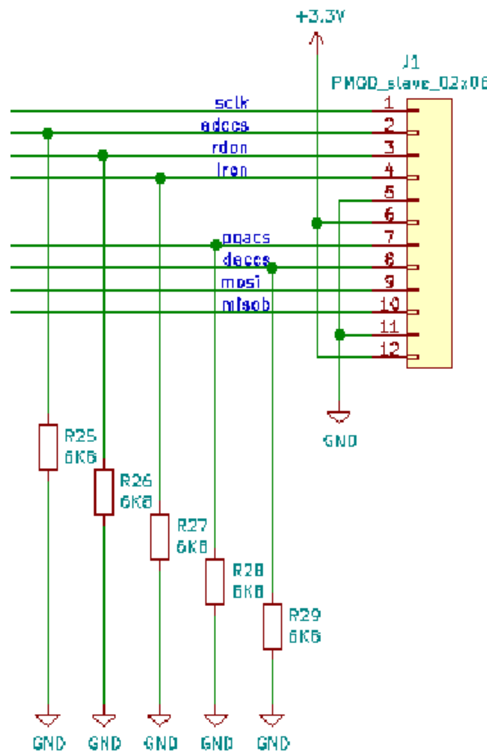


Figure 1.13: PMOD connector

One industrial standard PMOD connector takes all the signals for the oximeter interface

1	sclk	(input)	SPI sclk
2	adccs	(input)	chip select/sync for ADC – *active high*
3	redon	(input)	red led on
4	iron	(input)	infrared led on
5	GND	(pwr)	
6	Vss	(pwr)	3.3V
7	pgacs	(input)	chip select/sync for ADC – *active high*
8	daccs	(input)	chip select/sync for ADC – *active high*
9	mosi	(input)	SPI sdi (sdo for FPGA)
10	miso	(output)	SPI dso (sdi for FPGA)
11	GND	(pwr)	
12	Vss	(pwr)	3.3V

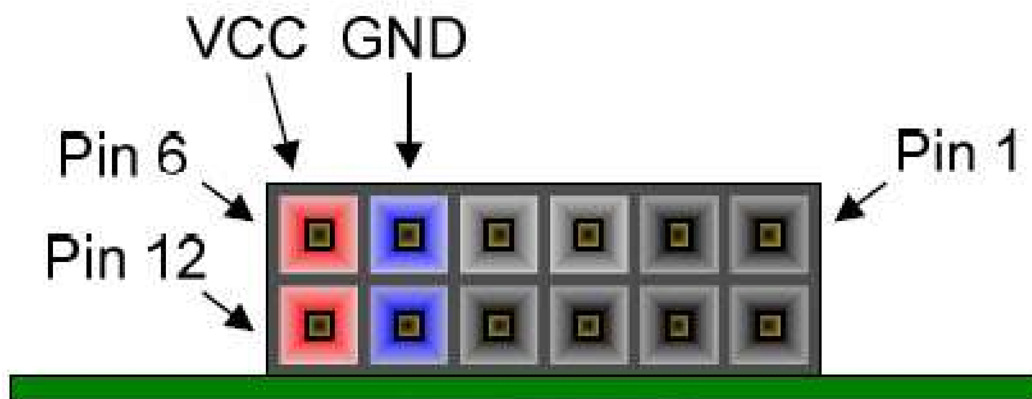


Figure 1.14: PMOD connection (front view of FPGA board)

The resistors (pull-down resistors) connected to GND ensure that all corresponding signals are inactive while the FPGA has not been configured (outputs passive).

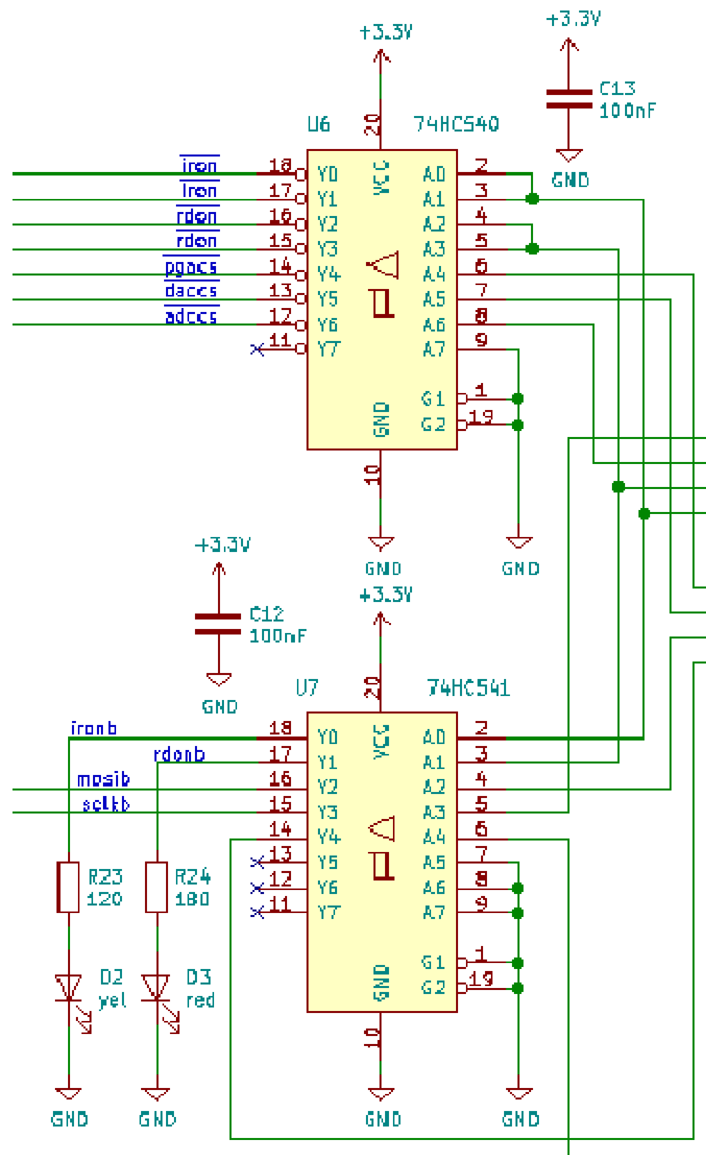


Figure 1.15: I/O buffer

The driver ICs U6 and U7 decrease the load of the FPGA pins. Visual copies of the red and infrared led signal are provided with LED1 and LED2 (especially useful for the infrared led in the finger clip).

The drivers provide enough current to charge and discharge the gates for the power MOSFETS Q2x-Q3x.

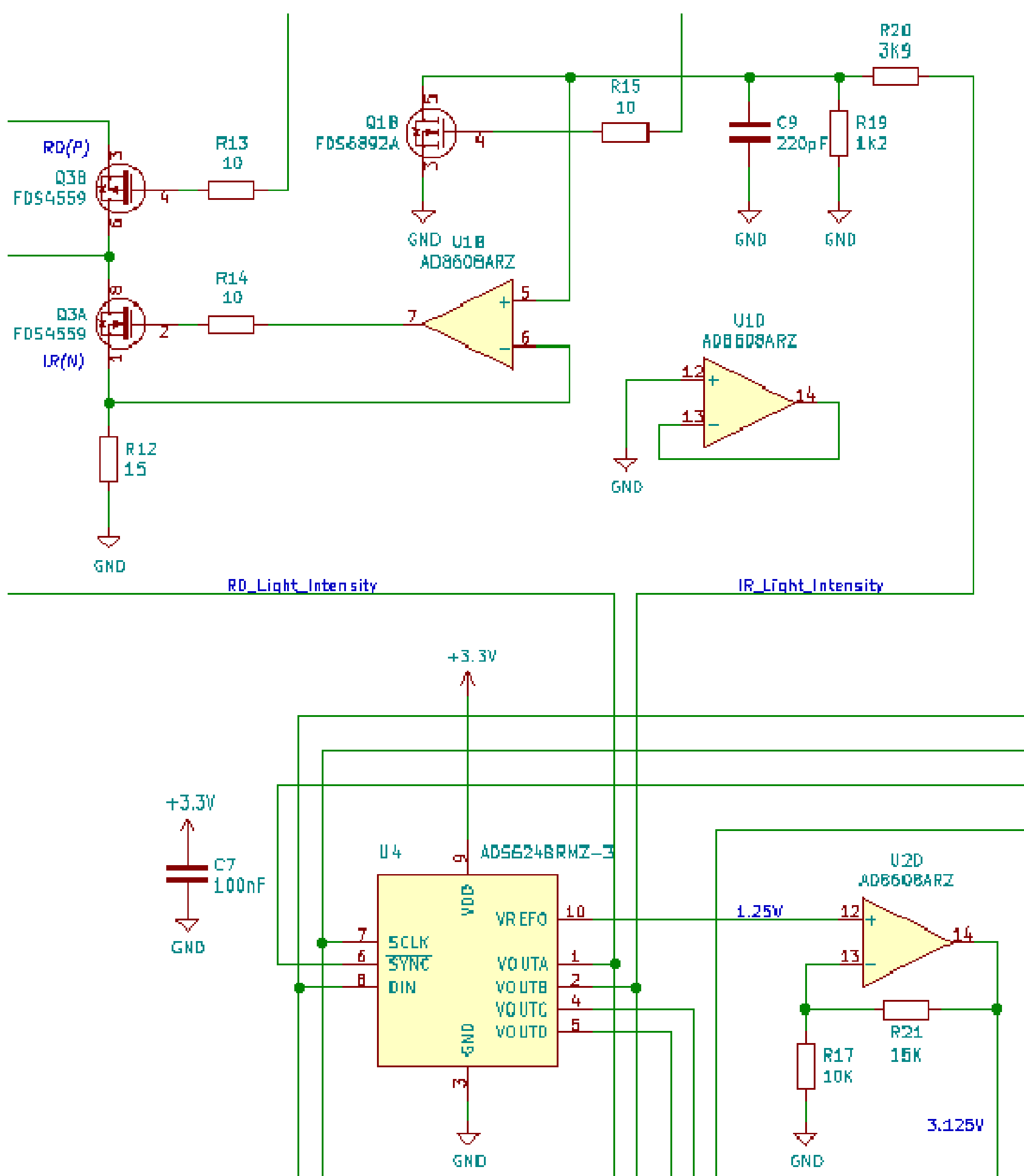


Figure 1.16: DAC and anti saturation circuit with voltage divider

The DAC provided for outputs VOUTA-VOUTD:

1	VOUTA	infrared light intensity
2	VOUTB	red led intensity
3	VOUTC	variable light indensity pulse indicator (LED)
4	VOUTD	ambient light cancelling

All outputs of the DAC are intended to be changed only if necessary, i.e. if the ambient light conditions change. The LEDs should not be switched on and off be the DAC. This is achieved be the digital signals RED and IR (see PMOD connector).

The DAC provides output (0V...2.5V). The voltage divider given by R19 and R20 (on on schematic view)) reduce the maximal output to

$$\frac{R_{19}}{R_{19} + R_{20}} \cdot \; 2.5 \text{ V} = 0.5882 \text{ V} . \tag{1.1}$$

The MOSFET Q3NA sets the output to zero volts when IR led is off. This prevents saturation for the operational amplifier IC3B. The DAC output can remain constant due to resistor R20.

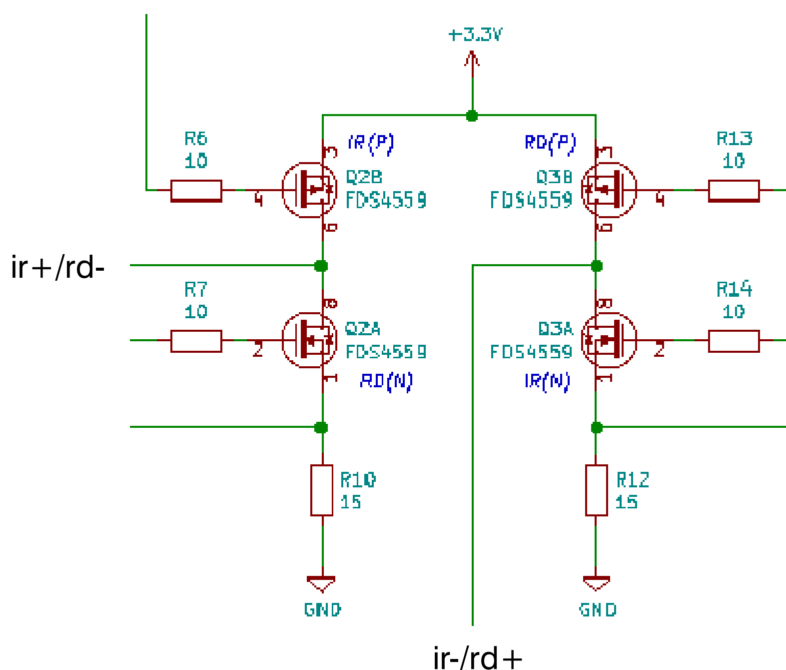


Figure 1.17: MOSFET H bridge with intensity control

The H bridge provides the output for the IR and RED LEDs. The led have only two terminals (anti-parallel connection, see block diagram fig. 1.12). Q2B (p channel) and Q3A (n channel) are responsible for the IR led. Q3B (p channel) and Q2A (n channel) are responsible for the RED led.

The OP U1B is a current controller for IR led. The voltage on terminal 6 will become identical to the voltage on terminal 5 (virtual connection). The current through IR led therefore becomes

$$\frac{\text{voltage on terminal 5}}{R_{12}} \; . \tag{1.2}$$

Exactly the same circuit exists for the IR led (not shown here).

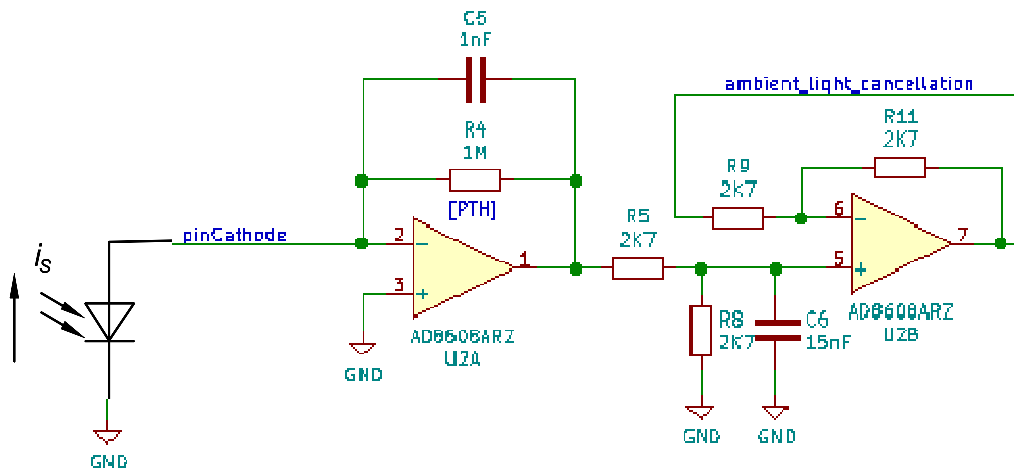


Figure 1.18: TIA with ambient light compensation (subtractor)

Red of infrared light causes a small current i_s in the reverse direction of the pin diode of the finger clip. This current is converted into a voltage at the output of U2A. The current to voltage conversion is called TIA (trans impedance amplifier). In the final version it might be required to change the value of R4 if the output voltage is too high or too low. A high quality operational amplifier (AD8608) is required here.

The next OP U2B subtracts the voltage VOUTD from the DAC to cancel ambient light. R9 could be increased to decrease the gain for the DAC voltage. Since all resistors R5, R8, R9 and R11 are equal the gain of the instrumentation amplifier is equal to one.

The components R5, R8 and C6 form a passive low pass according to

$$H(j\omega) = \frac{R_8}{R_5 + R_8} \frac{1}{1 + j\omega \frac{R_5 R_8}{R_5 + R_8} C_6} \tag{1.3}$$

This acts for as an anti-aliasing filter for the DAC. The cut-off frequency can easily made smaller with greater values for C6.

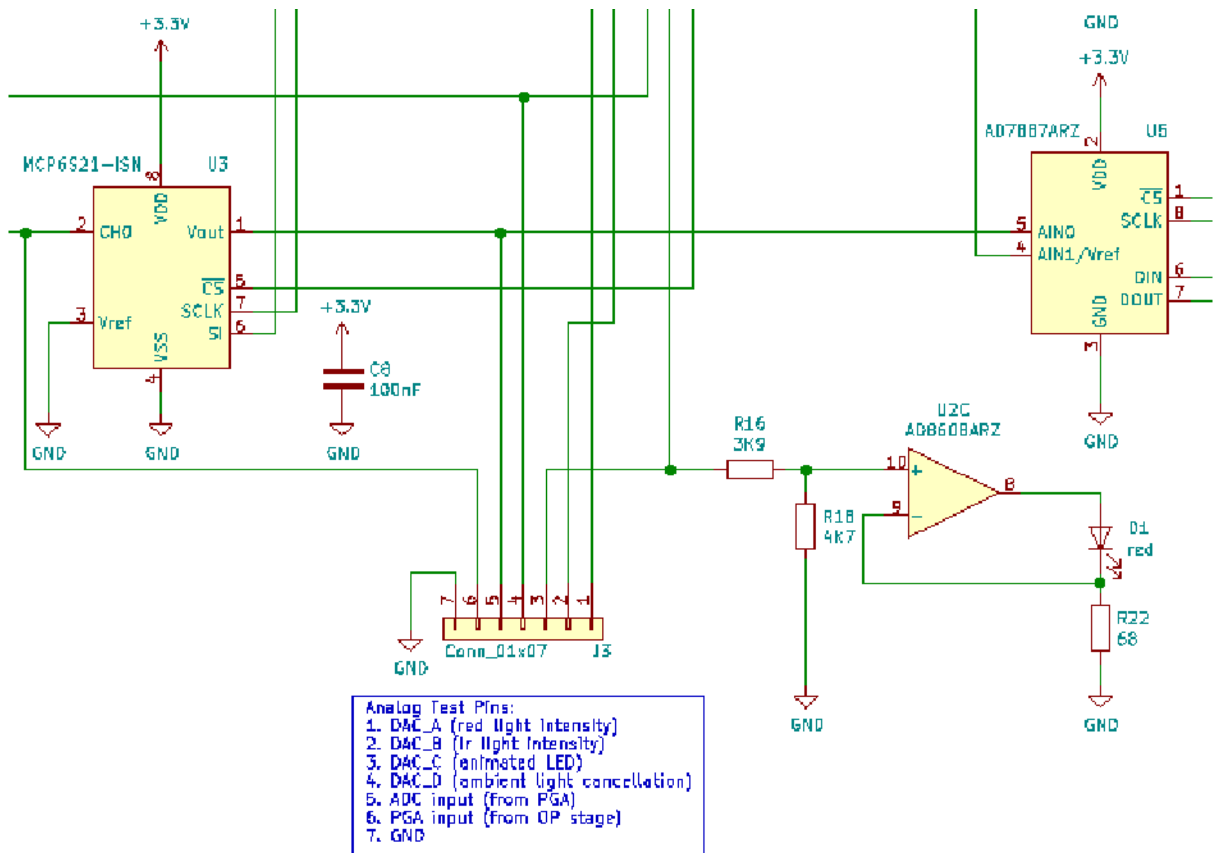


Figure 1.19: PGA with ADC section, analog test points

The SPI programmable variable gain amplifier (PGA) amplifies the sensor voltage for optimal resolution for AD conversion by the 12 bit ADC. Possible gains are 1, 2, 4, 5, 8, 10, 16 or 32.

Analog test points are provided for debugging purposes. Otherwise it is too difficult to make measurements on SMD pins of DAC, ADC, PGA and operational amplifiers.

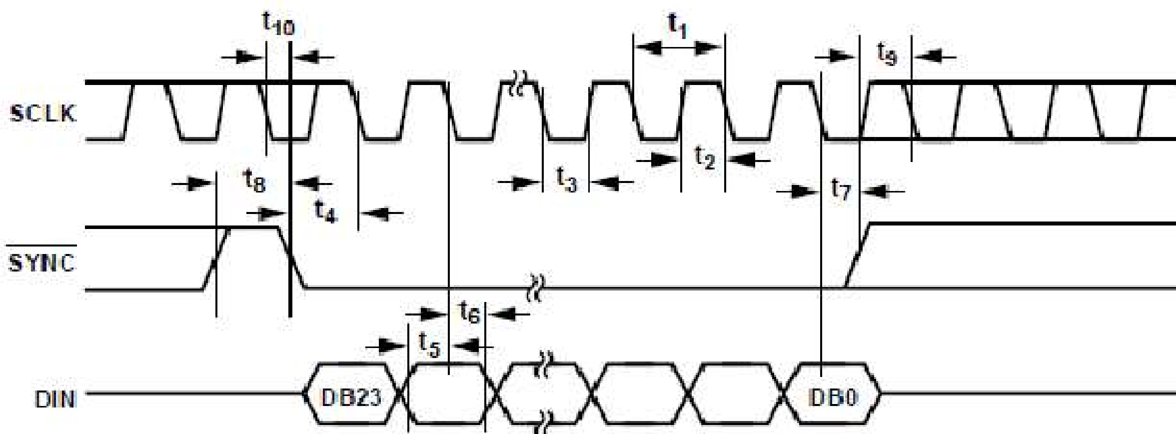


Figure 1.20: DAC SPI write cycle

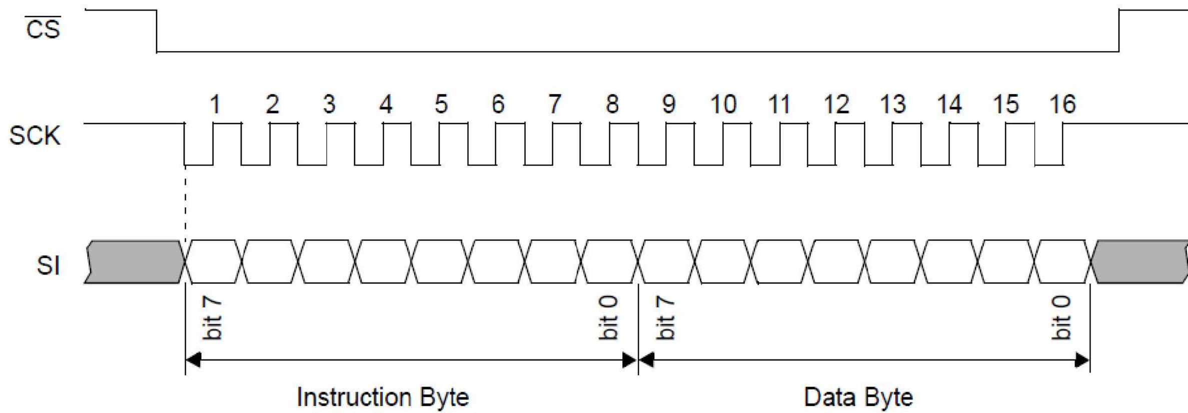


Figure 1.21: PGA SPI write cycle

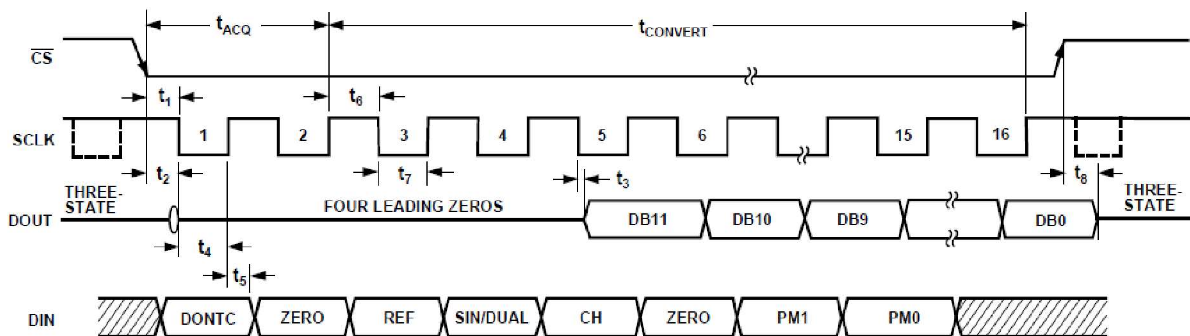


Figure 1.22: ADC SPI read/write

PGA, DAC and ADC share the same SPI lines (sclk, sdi, sdo). The transfers are shown in fig. 1.20 to fig. 1.22. Note that the length of the transfers differ for the devices.

3 Magnetic Levitation Model

A ferromagnetic ball should be positioned in a magnetic field. The magnetic force compensates the gravitational force acting on the ball.

The magnetic flux and the magnetic energy changes with the air gap $d-\delta$. By conservation of energy this must result in a change of mechanical energy and therefore results in a force F_m.

If the voltage u is the input to the system we obtain two degree of freedoms (current i and position x. If the current i becomes the input to the system (because a fast current controller exists) we have only δ or x as one degree of freedom.

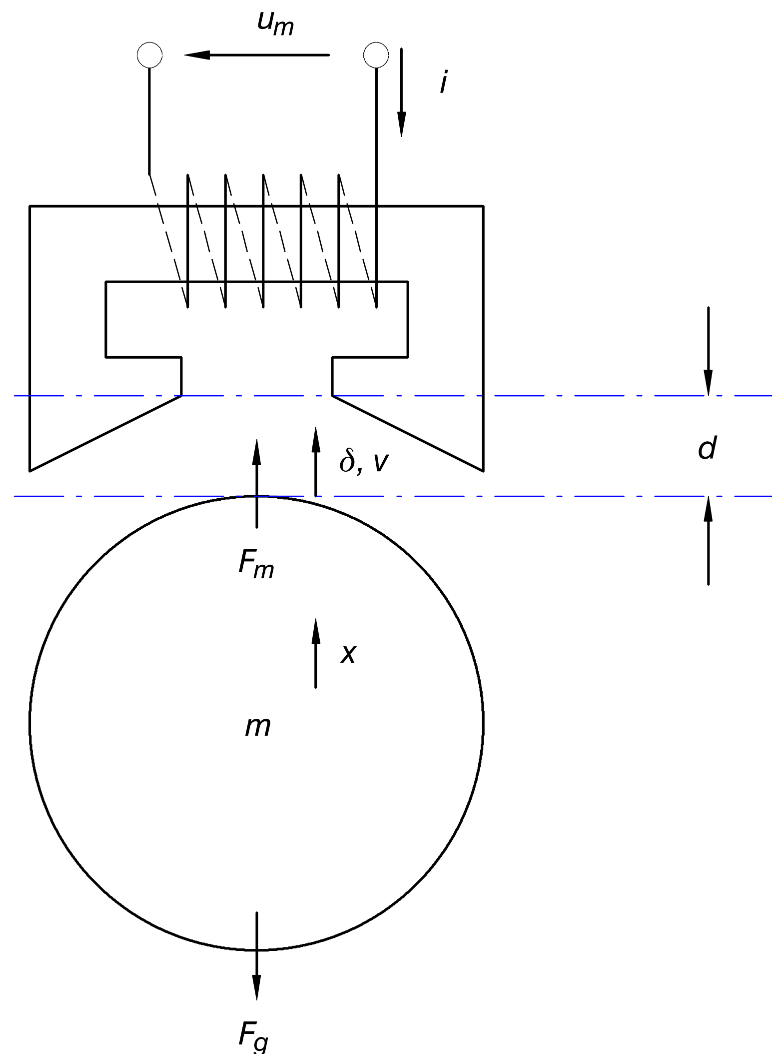


Figure 1.23: Magnetic levitation system

We will derive a simplified model assuming a constant magnetic field in the air gap between the magnet poles and the ball.

All equations of motion and the electrical equations can be completely obtained from the so-called *extended* Lagrangian energy L^{ex}. Therefore the magnetic coenergy, the kinetic coenergy and the potential energy are required.

If we assume that the magnetic field is closed over the ball the magnetic field becomes

$$H = \frac{ni}{2\,(d - \delta)} \tag{1.4}$$

or

$$B = \mu_0 H = \frac{\mu_0\, ni}{2\,(d - \delta)}\ . \tag{1.5}$$

With A being the active area for B we obtain for the flux linkage

$$\Psi = n\phi = nBA = \frac{\mu_0 A n^2 i}{2\,(d - \delta)}\ . \tag{1.6}$$

The magnetic energy (coenergy) becomes

$$W_m^i = \int_0^i \Psi(\delta, i')di' = \frac{\mu_0 A n^2}{4\,(d - \delta)}\, i^2 = \frac{k}{2}\,\frac{i^2}{d - \delta}\ . \tag{1.7}$$

The constant k is

$$k = \frac{\mu_0 A n^2}{2}\ . \tag{1.8}$$

The extended Lagrangian is (T is kinetic energy and V is potential energy)

$$L^{ex} = W_m^i + T - V = \frac{k}{2}\,\frac{i^2}{d - \delta} + \frac{1}{2} m v^2 - m g \delta\ . \tag{1.9}$$

The static terminals (resistors, voltage source, current source) are responsible for power supply and dissipative power. The effect of all static terminals are taken into account by the terms P^i and P^u satisfying the following relations

$$\frac{\partial P^u}{\partial u} = i\,, \qquad \frac{\partial P^i}{\partial i} = u\,, \qquad P^u + P^i = ui \tag{1.10}$$

for every single static terminal. The integrals

$$P^u = \int_0^u i(u')du' \quad \text{and} \quad P^i = \int_0^i u(i')di' \tag{1.11}$$

clearly are the solutions to (1.10). They also allow proper modelling of non-linear devices. For a loop with voltage supply (voltage u_m) and R (ohmic resistance of inductance) we obtain

$$P^i = \frac{1}{2} R i^2 - u_m i\,, \qquad P^u = \frac{u_R^2}{2R}\ . \tag{1.12}$$

Remarks: negative sign occurs when u and i have different directions, P^u of a voltage source is zero.

The electrical “equation of motion” becomes (P^u is not required)

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial W_m^i}{\partial i} + \frac{\partial P^i}{\partial i} = \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{ki}{d - \delta}\right) + Ri - u_m = 0\ . \tag{1.13}$$

The total derivative for t results in

$$\frac{k}{d - \delta}\,\frac{\mathrm{d}i}{\mathrm{d}t} + \frac{ki}{(d - \delta)^2} v + Ri = u_m\ . \tag{1.14}$$

Here the (trivial) equation

$$\frac{\mathrm{d}\delta}{\mathrm{d}t} = v \tag{1.15}$$

was used. The mechanical part is given by

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial L^{ex}}{\partial v}\right) - \frac{\partial L^{ex}}{\partial \delta} = 0\,. \tag{1.16}$$

With (1.9) we get

$$m\frac{\mathrm{d}v}{\mathrm{d}t} - \frac{k}{2}\frac{i^2}{(d-\delta)^2} + mg = 0\,. \tag{1.17}$$

Summing up all equations of motion give:

$$m\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{k}{2}\frac{i^2}{(d-\delta)^2} - mg\,, \tag{1.18}$$

$$\frac{\mathrm{d}\delta}{\mathrm{d}t} = v\,, \tag{1.19}$$

$$\frac{k}{d-\delta}\frac{\mathrm{d}i}{\mathrm{d}t} = u_m - \frac{ki}{(d-\delta)^2}v - Ri\,. \tag{1.20}$$

We can find an equilibrium state when all derivatives are zero in (1.18)−(1.20):

$$i_0 = \left(d-\delta_0\right)\sqrt{\frac{2mg}{k}}\,, \tag{1.21}$$

$$v_0 = 0\,, \tag{1.22}$$

$$u_{m0} = \frac{i_0}{R}\,. \tag{1.23}$$

Defining new variables as deviations from the equilibrium point

$$i := i_0 + u\,, \tag{1.24}$$

$$\delta := \delta_0 + x \tag{1.25}$$

result in

$$m\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{k}{2}\frac{\left(i_0+u\right)^2}{\left(d-\delta_0-x\right)^2} - mg\,, \tag{1.26}$$

$$\frac{\mathrm{d}\left(\delta_0 + x\right)}{\mathrm{d}t} = \frac{\mathrm{d}x}{\mathrm{d}t} = v\,. \tag{1.27}$$

With the selection of δ_0 to be in center position, i.e.

$$\delta_0 = \frac{d}{2} \tag{1.28}$$

we obtain with (1.21)

$$i_0 = \frac{d}{2}\sqrt{\frac{2mg}{k}} = d\sqrt{\frac{mg}{2k}}\ , \tag{1.29}$$

$$m\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{k}{2}\frac{\left(d\sqrt{\frac{mg}{2k}} + u\right)^2}{\left(\frac{d}{2} - x\right)^2} - mg\ . \tag{1.30}$$

It is easy to see that $u = 0$ and $x = 0$ no acceleration ($\mathrm{d}v / \mathrm{d}t$) occurs.

The linearized system at $x = 0$ and $u = 0$ gives

$$m\frac{\mathrm{d}v}{\mathrm{d}t} \approx 4\frac{mg}{d}x + \frac{\sqrt{8mgk}}{d}u\ . \tag{1.31}$$

or

$$\frac{\mathrm{d}v}{\mathrm{d}t} \approx 4\frac{g}{d}x + \frac{1}{d}\sqrt{8\frac{gk}{m}}\,u\ . \tag{1.32}$$

In state space we obtain with the state vector

$$\begin{bmatrix} v \\ x \end{bmatrix} \tag{1.33}$$

$$A = \begin{bmatrix} 0 & 4\frac{g}{d} \\ 1 & 0 \end{bmatrix}, \tag{1.34}$$

$$B = \begin{bmatrix} \frac{1}{d}\sqrt{8\frac{gk}{m}} \\ 0 \end{bmatrix}. \tag{1.35}$$

The linear state-space model can be used for designing a state feedback controller.

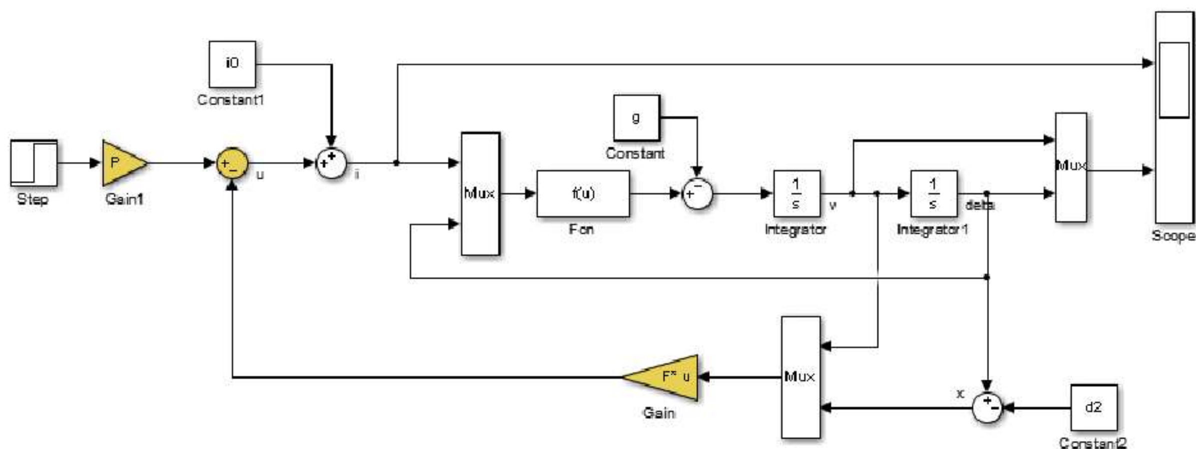


Figure 1.24: Possible simulation structure for maglev system and control

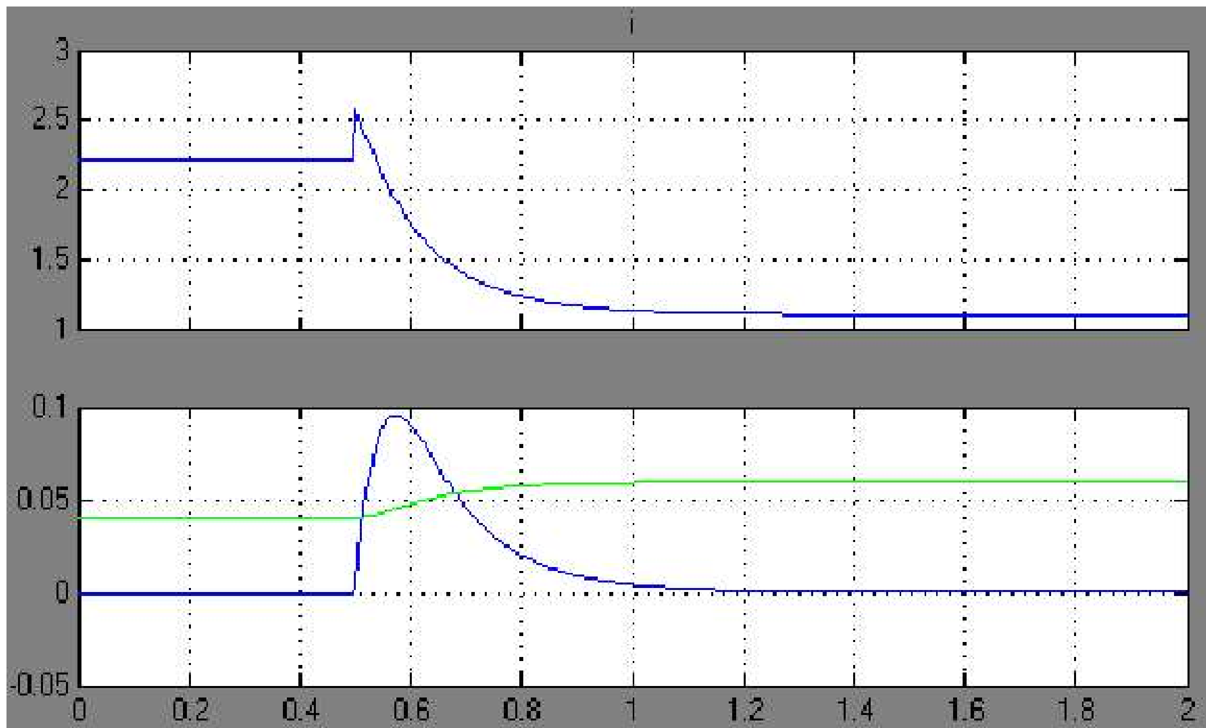


Figure 1.25: Position change (increase of x by 0.02, green), speed v is the blue curve, current i is shown in upper diagram

4 Electrical Interface to the Cart-Pendulum Experiment

Process interface signal should not be connected directly to the FPGA. In case the voltage level does not match a level conversion is required. For safety considerations any device that handles power should be isolated from the logic.

Modern logic operates at 3.3V or below. The standard logic for interfacing is still 5V. The HCPL-9030A combines voltage level conversion and digital isolation in the same device. Different supply voltages can be connected to both sides of the device.

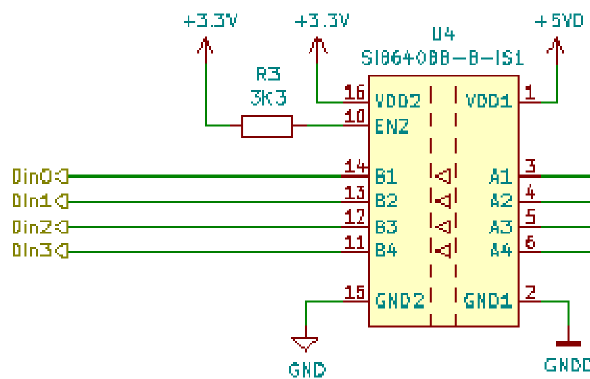


Figure 1.26: Voltage level conversion and digital isolation

Incremental encoder signals are often differential to avoid errors in transmitting signals in a noisy environment. The MAX3098 device perform differential to single-ended signal conversion. Since this device operates from Vcc = 5V, a level conversion according to fig. 1.26 is required.

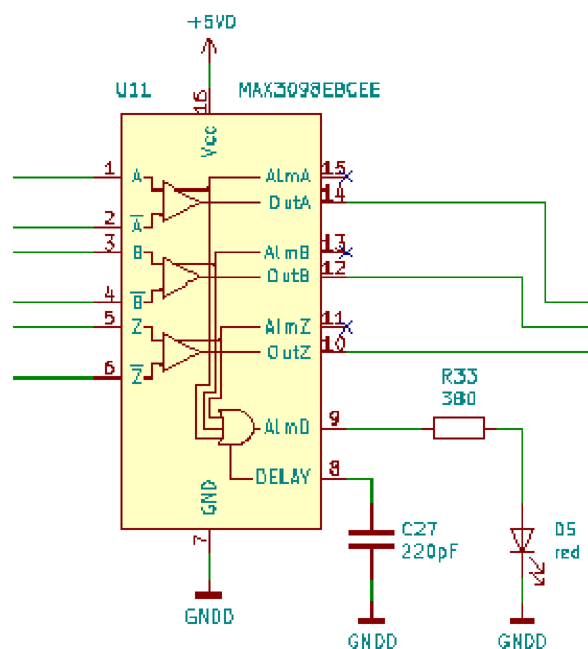


Figure 1.27: Differential signal / single ended voltage conversion

The DAC output in unipolar rail-to-rail while the process requires a bipolar analog input. The circuit in fig. 1.28 provides bias and suitable gain for this purpose.

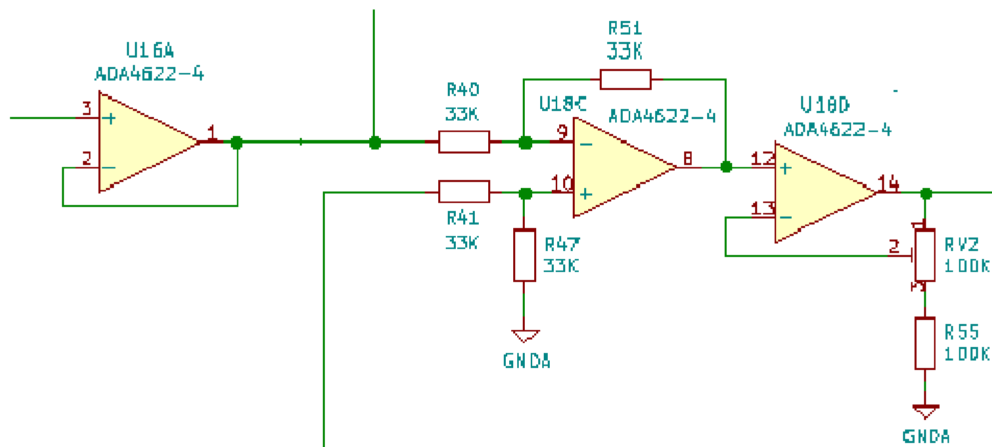


Figure 1.28: Analog level change (bias and gain)

The circuit provides gain and bias to convert an unipolar 0...2.5V input to ± 2.5V up to ± 5.0V (adjustable by RV2).

The first operational amplifier provides a low impedance 1.25V voltage reference (half of the full scale input voltage from top). The operational amplifier can be slow and must be unity gain stable, since the full gain is used in it's feedback loop.

The difference amplifier circuit with single resistor values is shown below.

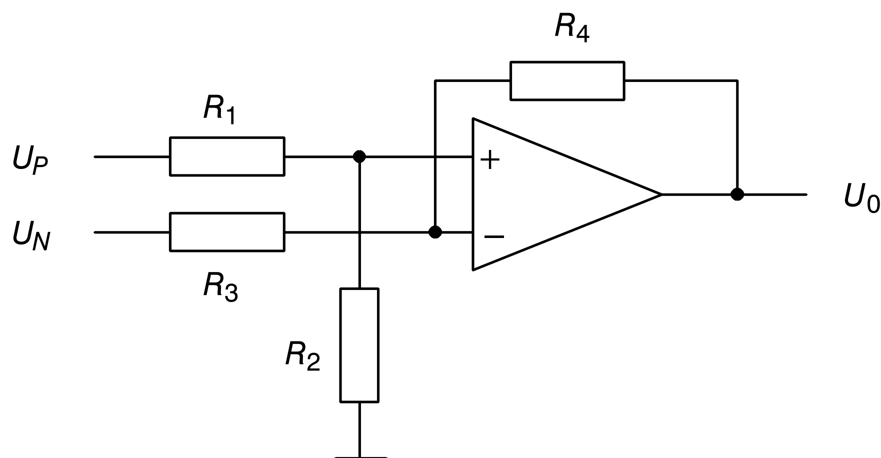


Figure 1.29: Difference amplifier circuit

Since circuit in fig. 1.29 is linear, it can be decomposed into two simpler circuits. The equations for the complete circuit is the sum of the equations for the two simpler circuits.

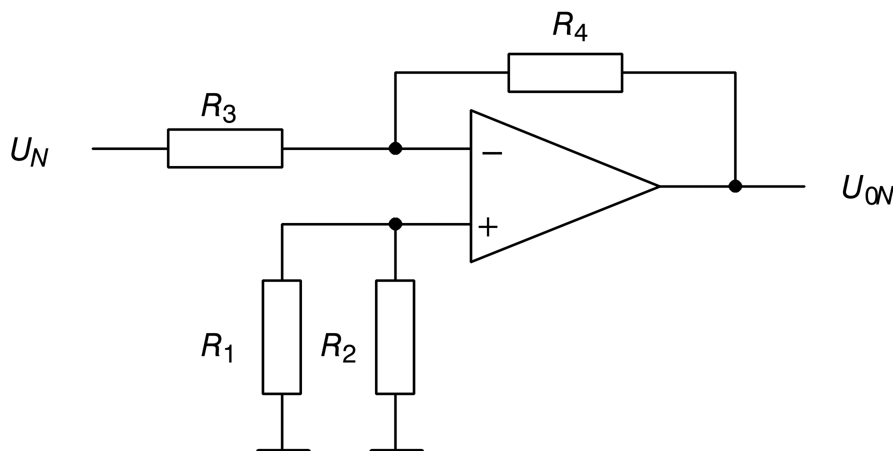


Figure 1.30: Difference amplifier circuit for $U_P = 0\text{V}$

According to fig. 1.30 the output becomes

$$U_{0N} = - \frac{R_4}{R_3} U_N . \tag{1.36}$$

The resistors R_1 and R_2 have no effect.

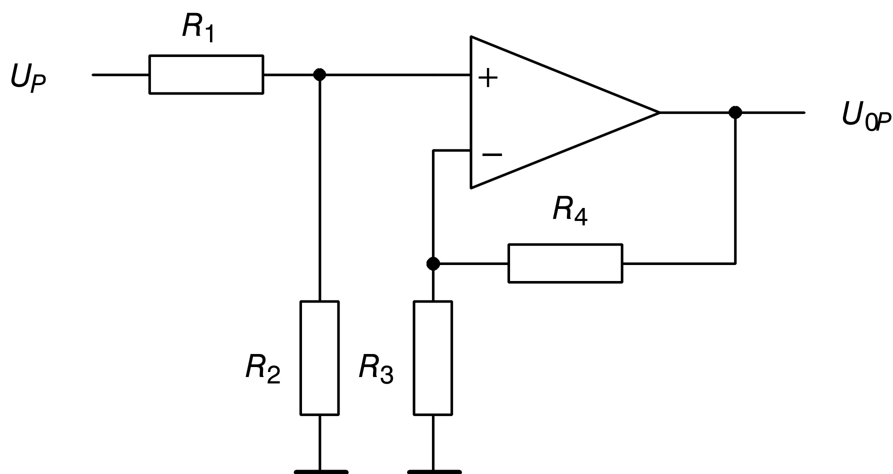


Figure 1.31: Difference amplifier circuit for $U_N = 0\text{V}$

The non-inverting amplifier has a gain of $R_3 / (R_3 + R_4)$. The input signal passes a voltage divider given by R_1 and R_2. This results in

$$U_{0P} = \frac{R_2}{R_1 + R_2} \frac{R_3 + R_4}{R_3} U_P . \tag{1.37}$$

The sum of the two results is the total output voltage

$$U_0 = U_{0N} + U_{0P} = \frac{R_2}{R_1 + R_2} \frac{R_3 + R_4}{R_3} U_P - \frac{R_4}{R_3} U_{0P} . \tag{1.38}$$

If we set

$$R_2 = \alpha R_1 \qquad \text{and} \qquad R_4 = \alpha R_3\,, \tag{1.39}$$

we obtain

$$U_0 = \frac{\alpha R_1}{(1+\alpha)R_1} \frac{(1+\alpha)R_3}{R_3} U_P - \frac{\alpha R_3}{R_3} U_{0P} = \alpha\left(U_{0P} - U_{0N}\right). \tag{1.40}$$

Care has to be taken to prevent used OP components from “ringing”. The following method disables safely unused operational amplifiers. The OPs must be unity stable to use this circuit, i.e. the phase between input and output should not exceed 90 degrees for any gain.

Figure 1.32: Disabling unused op-amps

The complete interface schematic is shown below.

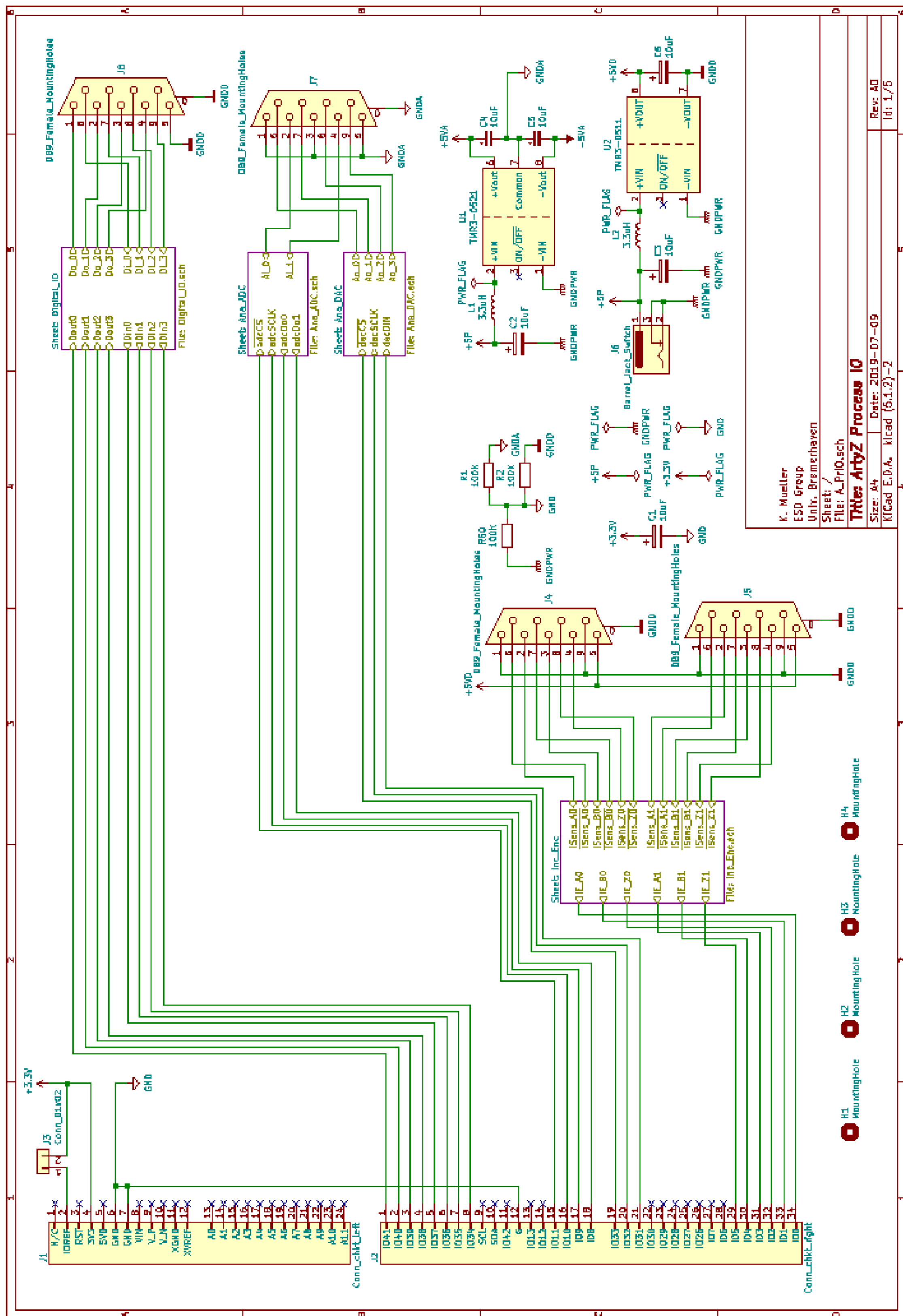


Figure 1.33: Complete schematic (top level)

4.1 I/O Signals

The communication with the process requires digital I/O, digital sensor input and analog output. The digital signals are level shifted to + 5V TTL. The digital signals are as follows:

Name	Direction	Meaning
switch_left	in	end switch left side
switch_right	in	end switch right side
amp_enable	out	enable servo amplifier
amp_ok	in	servo amplifier operational

The sensor inputs are listed below. The cart sensor has differential outputs while the pendulum angle is single-ended.

Name	Direction	Meaning
cartA+	in	cart sensor track A positive
cartA-	in	cart sensor track A negative
cartB+	in	cart sensor track B positive
cartB-	in	cart sensor track B negative
pendA	in	pendulum angle sensor track A
pendB	in	pendulum angle sensor track B

The servo amplifier receives an analog signal. The motor torque is proportional to the signal voltage.

Name	Direction	Meaning
mforce	out	±3.75V for motor torque/cart force

4.1.1 Data Formats

The data format for integer data is crucial for operation. The selected data format should prevent overflow, underflow and should not cause a loss of precision.

The microcontroller provides native 32 bit integer operations. Any 32 bit operation is always executed with maximum speed since these operations are executed in hardware.

The DAC has 12 bits resolution. It can be left-aligned to 16 bits ignoring bits 12...15. If F_{max} is the maximum force the motor can generate the following equation allows conversion of internal data representation and physical values

$$\frac{F}{F_{\max}} = \frac{mforce}{2^{15}} \; . \tag{1.41}$$

This makes *mforce* an int_16.15 format number. The integer variable *mforce* can represent the physical range from $-F_{max}$ .... + 0.99997 F_{max}, which is almost the full range.

The pendulum angle is received from a 16 bit counter. The angle can be handled in the same way

$$\frac{\alpha}{\alpha_{\max}} = \frac{pangle}{2^{15}} . \tag{1.42}$$

The integer variable *pangle* is an int_16.15 number. It can be positive and negative. Full precision is preserved.

The cart is received from a 24 bit counter

$$\frac{x}{x_{\max}} = \frac{cartp}{2^{23}} . \tag{1.43}$$

The integer variable *cartp* is an int_24.23 number to provide full precision.

4.2 Process User Interface

The process will be controlled by a Java application program on a PC. The FPGA is connected by a serial line at 115200 bit/s. Since native support for serial lines was officially dropped fro JAVA the RXTXcomm (gnu.io) library was used to establish serial connections at any desired baud rate.

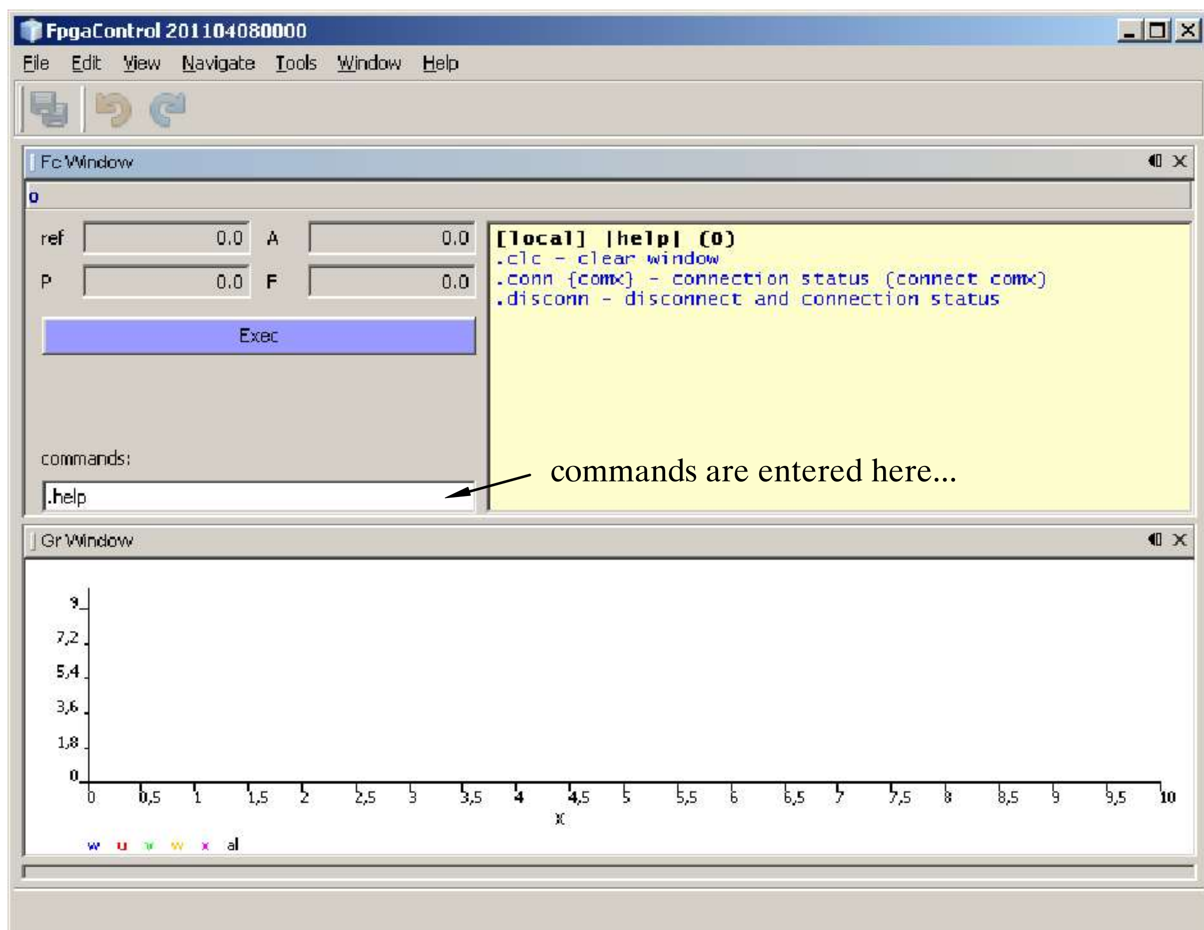


Figure 1.34: JAVA Netbeans platform GUI

The GUI consists of two windows which can be dragged and resized as required. In the command text field both internal commands (start with ".") and external commands (will be transferred to MicroBlaze) can be submitted. The commands must be extended to allow control operation of the cart-pendulum system.

4.2.1 Internal Commands

`.clc`

Clear command window (like Matlab's clc command)

`.dbg`

Display debug information about the state of the serial communication. This is sometimes useful when using not fully compatible USB serial adapter.

`.plot {filename}`

.plot without parameter plots the PC side data buffer into the diagram. If a file name is supplied it will create in addition a file (located under \Temp\....) with the same data. This file can be loaded into Matlab (e.g. "load measure_x.dat").

`.ppar {reset | min [k] | max [k] }`

Plot range parameter setting. "reset" plots the full range. With integers for "min" or "max" the range for plotting can be narrowed.

`.upd { on | off }`

Update the fields ref (reference value), A (angle), P (position) and F (force) every 500 ms.

`.conn { com_x }`

Connects to a serial port. Without parameters it will list available connections.

`.disconn`

Disconnect from a connected serial port.

4.2.2 External Commands

`exit`

Terminates the MicroBlaze main program

`mode { on | off }`

Set the interrupt mode variable "Intr_Phase".

`log { on | off | clr | show }`

Logger commands for start, stop, initialization and list.

`regs`

Shows register contents

`downl`

Download logger data (if available)

`ref [ value ]`

Sets the reference value for cart position

`dacch [ value ]`

Sets the DAC channel for “dacval” commands, $0 \leq$ value $\leq$ 3.

`dacval [ hexvalue ]`

Output of analog value “hexvalue” on channel specified by “dacch” command. 0000 $\leq$ hexvalue $\leq$ ffff.

`r`

Read and display position information form all position sensors.

`p`

Clear position counters.

`help`

Prints out an (incomplete) list of available commands.

4.3 Retrieving Process Data

In order do receive process data the logger need to switched on (”log on”). After Measurement was taken the logger has to switch off (”log off”). The command “downl” receives all data from a ring buffer.

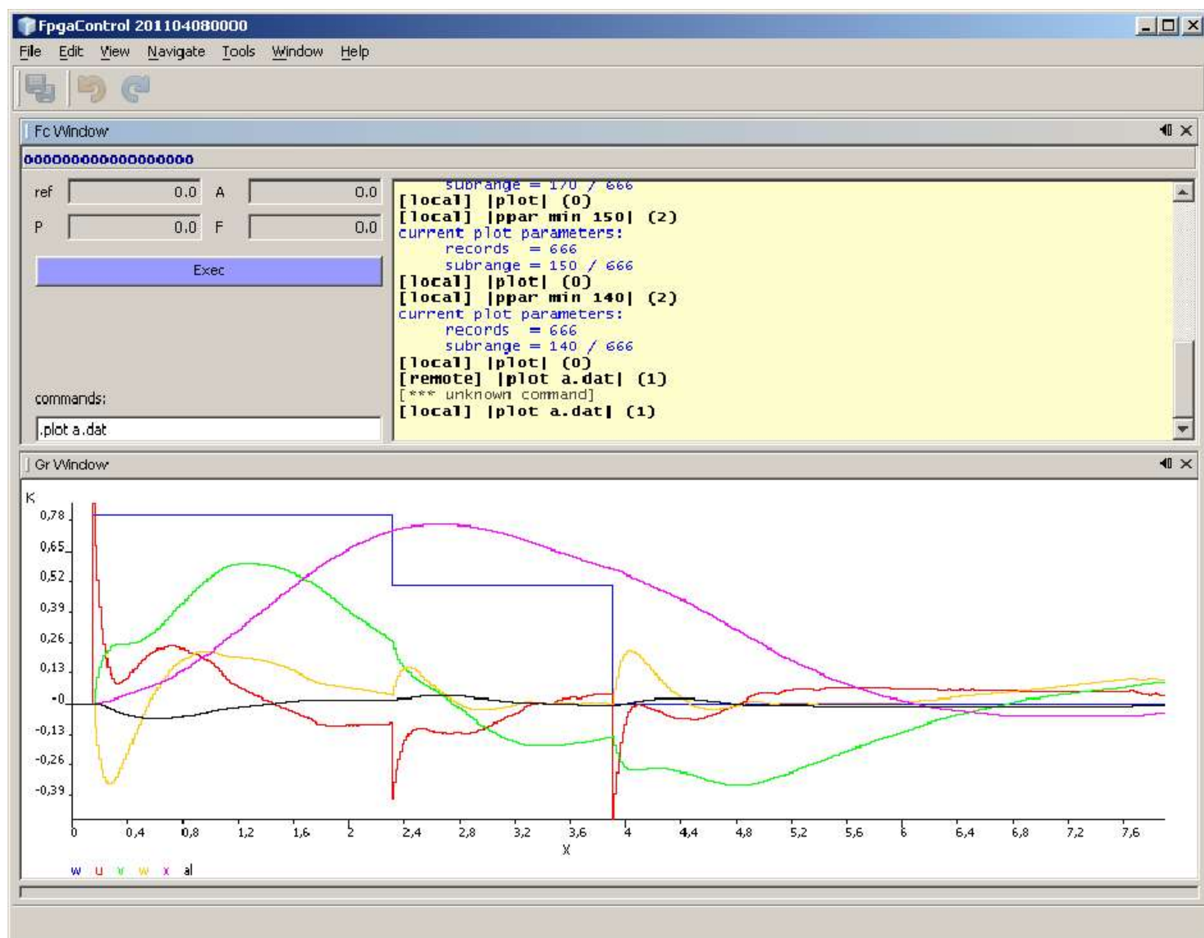


Figure 1.35: JAVA GUI after download and plot

The data can be loaded into Matlab for plotting and parameter identification purposes:

```
load a.dat
plot( a( :, 1) ,  a( :, 2 :end) )
```

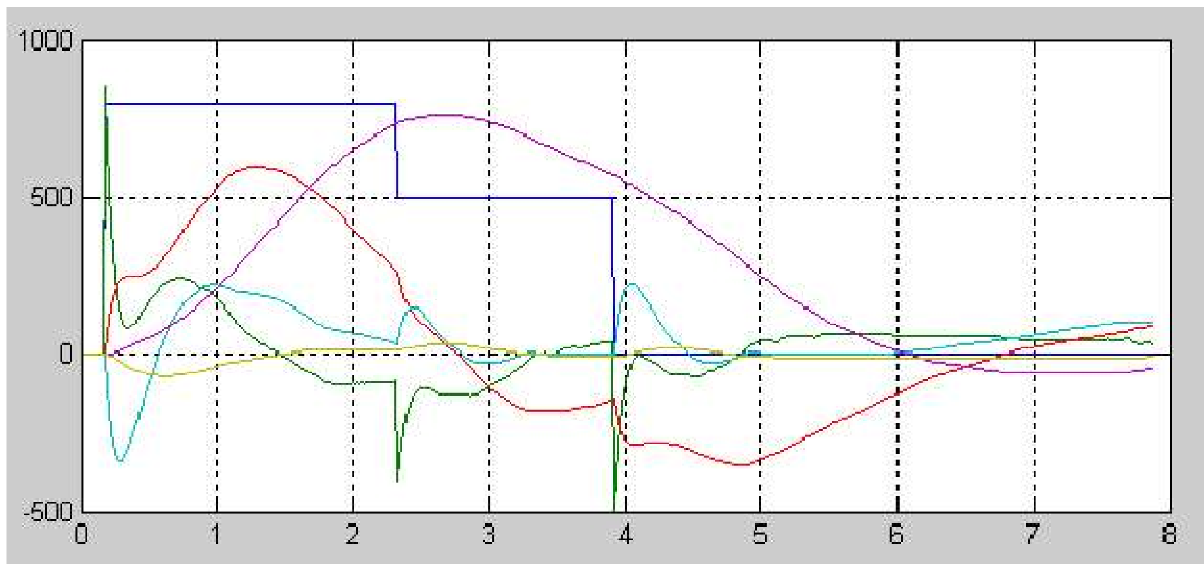


Figure 1.36: Data loaded into Matlab and the command plot(a(:,1), a(:,2:end))

5 PI Controller for the Initialization Phase

For finding the initial position (referencing) a PI type controller can be used to obtain a low speed for finding the reference switches (left and right).

The structure of a discrete PI controller is is shown in fig. 1.37.

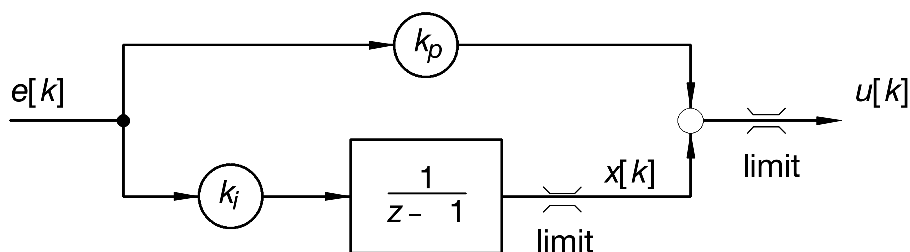


Figure 1.37: Discrete PI controller

Using the shift operator 1/z the integrator transfer function block diagram becomes

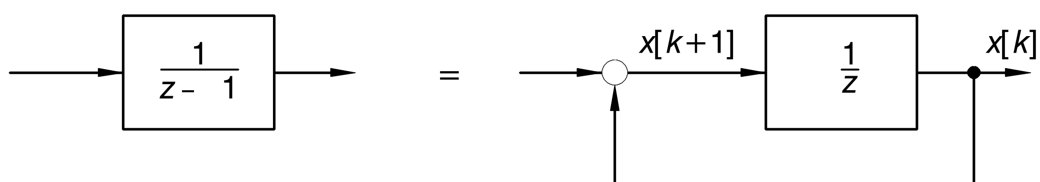


Figure 1.38: Detailed block diagram of a discrete integrator

The difference equation for the discrete PI controller are

$$u[k] = k_p e[k] + x[k] , \tag{1.44}$$

$$x[k+1] = k_i e[k] + x[k] . \tag{1.45}$$

It is important not no reverse the sequence of the computation when implementing the difference equations. The limit functions have been omitted for clarity.

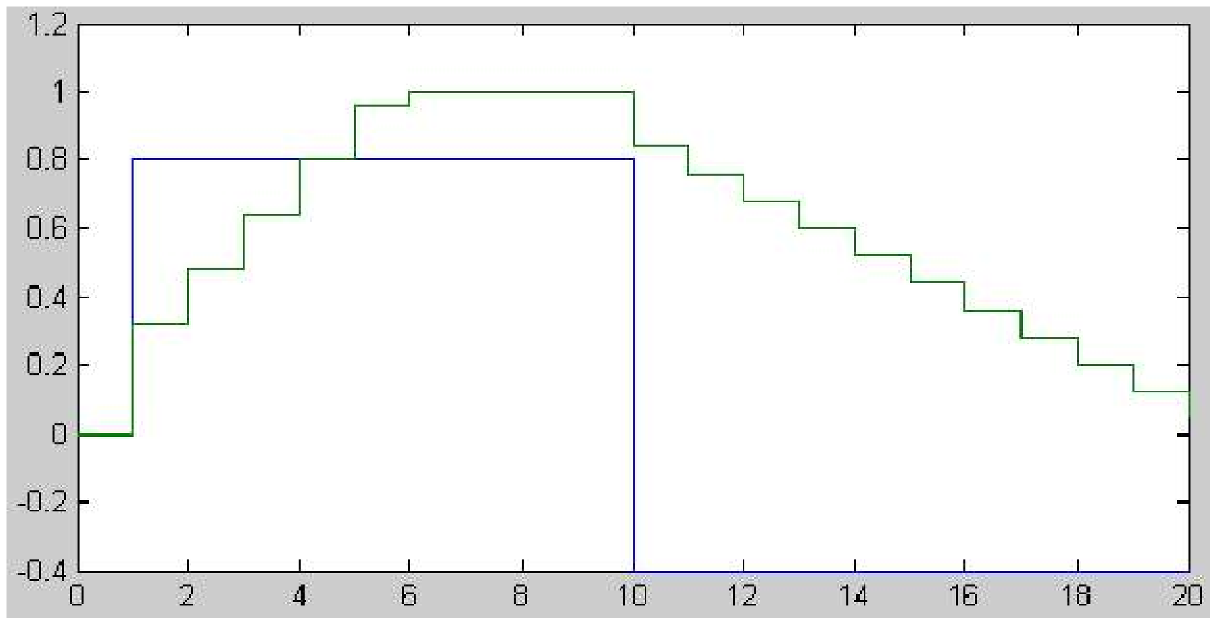


Figure 1.39: PI controller with limits on integrator and output

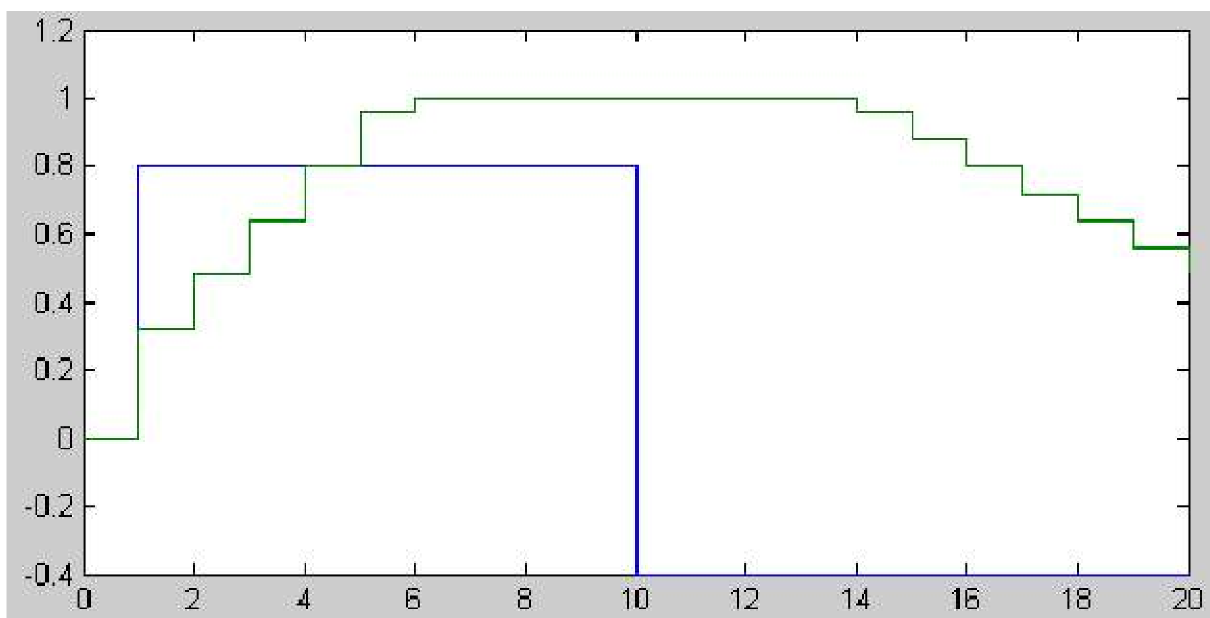


Figure 1.40: PI controller with limited output only

The effects of limiters are shown in figures 1.39 and 1.40. While PI controller with limiters on integrator and output reacts immediately on the input signal, the version with output only limiter keeps saturated for several sampling intervals.

5.1 Fixed Point PI Controller

The feature of C++ to create dynamic objects is critical in real time systems especially if memory is limited. Memory allocation and cleanup makes such a system unpredictable. The system may also run out of memory due to changing memory demand during execution. In C++ a PI controller might look similar to:

```
class PIctrl {
    private:
        int xi;
        int kp, ki, limit;
    public:
        Pictrl(int pgain, int igain, int limit);
        ~PIctrl();
        .
        .
        .
};
```

The benefits are data encapsulation and a clean collection of methods for that object. The biggest advantage of this structure is *reusability.*

However, the same same policy can be applied (and should be applied) to ANSI C programs although C does not require to program in this way.

It is important to know the data formats required for computation. Since IO data requires only 16 bits it is sufficient to provide this for input and output variables. In addition the result of a 16 bit x 16 bit multiplication fits safely into 32 bit without overflow. In the following detailed block diagram includes suitable data formats.

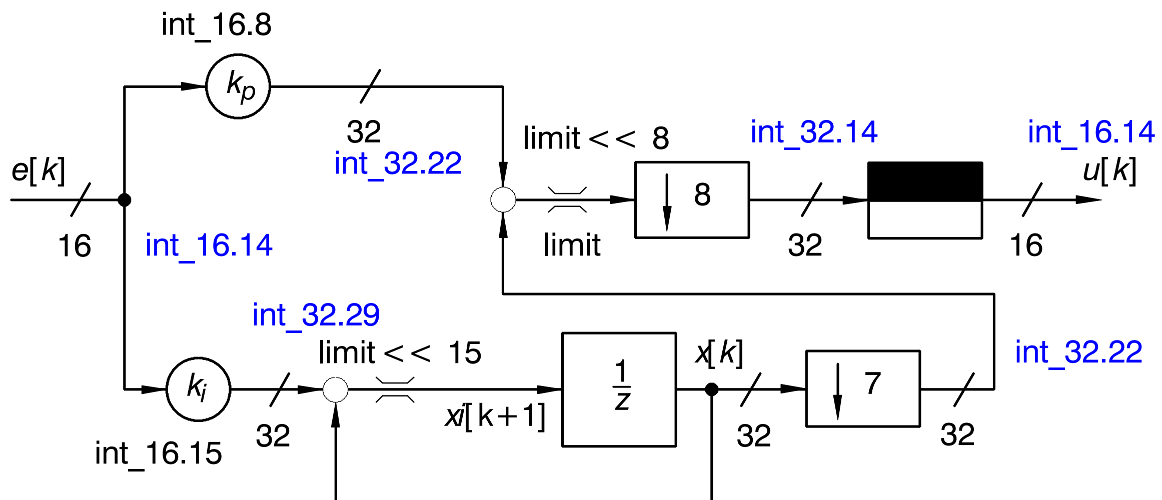


Figure 1.41: Discrete PI controller data formats (suitable for 32 bit µC)

The data formats for k_p (may be less or greater 1) is selected to int_16.8 so the proportional gain is in the range of (int_n.f: $-2^{n-1}/2^f .. + 2^{n-1} - 1/2^f$)

$$\text{int_16.8 :} \quad -128.0 \leq k_p \leq +127.996 . \tag{1.46}$$

Since the intergal gain k_i is usually less than 1 the format is

$$\text{int_16.15 :} \quad -1.0 \leq k_i \leq +0.99997 . \tag{1.47}$$

The data types for singed (**s**xx) and unsigned integers (**u**xx) for a 32 bit microprocessor can be defined in a header file:

```
#ifndef XIL_TYPES_H_
#define XIL_TYPES_H_

/** Constant Definitions **/

#ifndef TRUE
#  define TRUE  1
#endif

#ifndef FALSE
#  define FALSE  0
#endif

#ifndef NULL
#define NULL  0
#endif

typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned long u32;
```

```
typedef unsigned long long u64;
typedef char s8;
typedef short s16;
typedef long s32;
typedef long long s64;

#endif /* XIL_TYPES_H_ */
```

It should be verified that the architecture dependent data types have the correct size:

```
printf(”Verifying data type definitions:\n”);
printf(”  8-Bit:  u8/s8  (%d/%d)\n”,
        sizeof(u8), sizeof(s8));
printf(” 16-Bit: u16/s16 (%d/%d)\n”,
        sizeof(u16), sizeof(s16));
printf(” 32-Bit: u32/s32 (%d/%d)\n”,
        sizeof(u32), sizeof(s32));
printf(” 64-Bit: u64/s64 (%d/%d)\n”,
        sizeof(u64), sizeof(s64));
```

The output should be as follows:

```
Verifying data type definitions:
  8-Bit:  u8/s8  (1/1)
 16-Bit: u16/s16 (2/2)
 32-Bit: u32/s32 (4/4)
 64-Bit: u64/s64 (8/8)
```

Now we can define the PI controller “object”:

```
#ifndef PI_CONTROL_H_
#define PI_CONTROL_H_

#define  KP_SHIFT  8          // int_16.8
#define  KI_SHIFT  15         // int_16.15

typedef struct {
        s16 kp;
        s16 ki;
        s32 limit;
        s32 limint;
        s32 xi;
} PiObj_struct;

void UPiCtrlInit(s16 pgain, s16 igain, s16 limit);
s16 UPiCtrlC(s16 ctr_e);
```

```
#endif /* PI_CONTROL_H_ */
```

Two “methods” are declared:

`UPiCtrlInit:` Initializing method
`UPiCtrlC:` Controller calculation

The controller can be implemented by creating a static object PiObj and the “methods”:

```
#include "xil_types.h"
#include "pi_control.h"

static PiObj_struct PiObj;    // the "object"

void UPiCtrlInit(s16 pgain, s16 igain, s16 limit) {
        PiObj.kp = pgain;
        PiObj.ki = igain;
        PiObj.limit = limit << KP_SHIFT;
        PiObj.limint = limit << KI_SHIFT;
        PiObj.xi = 0;
}

s16 UPiCtrlC(s16 ctr_e) {
    s32 u_ctr;
    u_ctr = PiObj.kp * ctr_e
          + (PiObj.xi >> (KI_SHIFT - KP_SHIFT));
    if (u_ctr > PiObj.limit)  u_ctr = PiObj.limit;
    else if (u_ctr < -PiObj.limit)  u_ctr = -PiObj.limit;
    u_ctr >>= KP_SHIFT;
    PiObj.xi += PiObj.ki * ctr_e;
    if (PiObj.xi > PiObj.limint)  PiObj.xi = PiObj.limint;
    else if (PiObj.xi < -PiObj.limint)
       PiObj.xi = -PiObj.limint;
    return (s16)u_ctr;
}
```

Note that the controller code is not mixed up with any application specific IO data.

6 C Code Verification in Simulink

The fixed point C code can be verified in Simulink by including a *legacy code block* in a Simulink model. The C code under investigation must reside in a module and a corresponding header file. The block creation is controlled by a legacy code description structure. Here is an example for a position controller C code block:

```
[01]  % create legacy mex function for cart control
[02]  def = legacy_code('initialize');
[03]  def.SourceFiles = {'posctrl.c'};
[04]  def.HeaderFiles = {'posctrl.h'};
[05]  def.SFunctionName = 'ex_sfun_posctrl';
[06]  % short posctrl(short reset, short ref, short x,
                     short v);
[07]  def.OutputFcnSpec = 'int16 y1 = posctrl(int16 u1,
                          int16 u2, int16 u3, int16 u4)'
[08]  legacy_code('sfcn_cmex_generate', def)
[09]  legacy_code('compile', def)
[10]  legacy_code('slblock_generate', def)
[11]  disp('That's all folks.')
```

[02]: Initializes the structure

[03]: List of source files. In this case we have only one source. It is common practice (although not required) that the function name to export has the same name as the source file.

[04]: List of header files

[05]: S-function name, the name for the compiled C source.

[06]: (comment), the C source function definition as from the header file. It has four 16 bit signed integer integer parameters (block inputs) and returns a 16 bit signed integer (output of the Simulink block).

[07]: Equivalent Simulink block input and output description. All inputs are denoted by u and all outputs are denoted as y. Please note, that Simulink has it's own type names.

[08]: Generate mex file command

[09]: Compiler invocation, if available Microsoft VC++ is used, otherwise LCC (GNU) is a possible alternative (but less efficient).

[10]: Create a Simulink block (for insertion into the Simulink model).

If the process runs successfully the following block becomes available.

Figure 1.42: Generated Simulink block from C code

The block can be inserted into a Simulink model but *data type conversion blocks* are required to access the other blocks of the model. In the following model the data format for fixed point arithmetic IO is int_16.14.

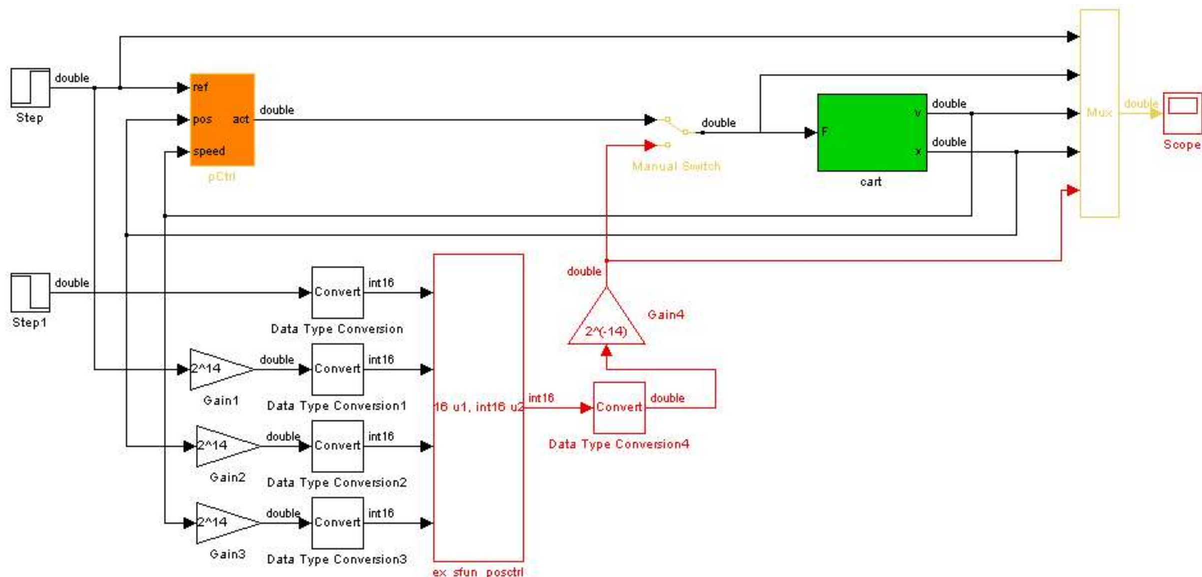


Figure 1.43: C code block in a Simulink model (including data type conversion blocks)

The detailed block diagram of the linearized process (the green block) is shown below.

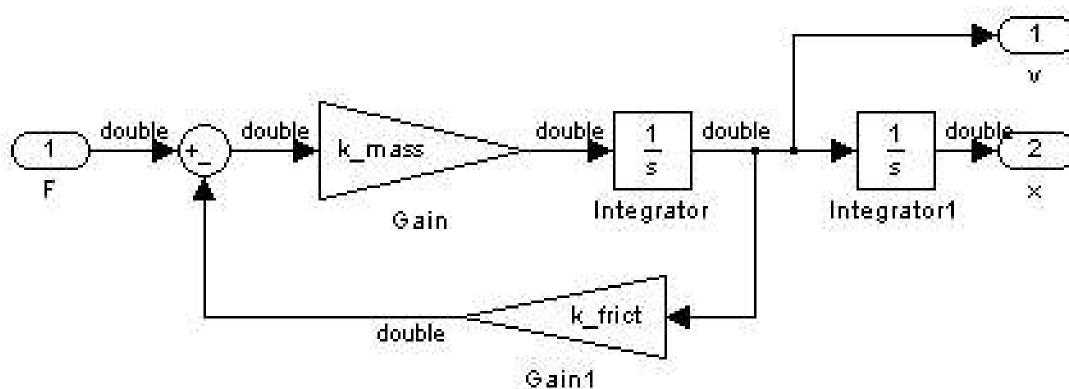


Figure 1.44: Process model

The yellow position control block is a standard position control cascade controller (without output limits to keep it simple). The speed PI controller is discrete.

The C code for the function that is exported as a block is shown below:

```
// position controller (discrete)
short posctrl(short reset, short ref, short x, short v) {
    short  uu;
```

```
        if (reset > 0) {
            UPiCtrlInit(&Speed_Ctrl, KP_V, KI_V, 0x7fff);
            UPiCtrlInit(&Pos_Ctrl, KP_X, 0, 0x7fff);
            uu = 0;
        } else {
            uu = UPiCtrlC(&Pos_Ctrl, ref-x);
            uu = UPiCtrlC(&Speed_Ctrl, uu-v);
        }
        return uu;
    }
```

If the reset signal is 1, the PI speed controller and the P position controller ($k_i = 0$) are initialized. If reset becomes zero, the cascade control is computed. Note that the controller structure is passed as a pointer to the controller object. The controller is almost the same as of section 5.1 (except pointer for the PiObj_struct).

- Create a Simulink model cartpctrl.
- Add a (linear) process model as of fig. 1.44.
- Add a FcnPreloadFunction cartpcrtl_init for initialization of the simulation variables.
- Create a discrete position controller and design the proportional and integral gains for position control (no optimal controller is required here).
- Write the posctrl.h header file and code posctrl.c in fixed point.
- Run cplegacy.m to create a Simulink block. Correct compilation errors if necessary.
- Run the whole simulation and compare the Simulink controller with the fixed point C code controller. The result should look similar to the diagram in fig. 1.45.

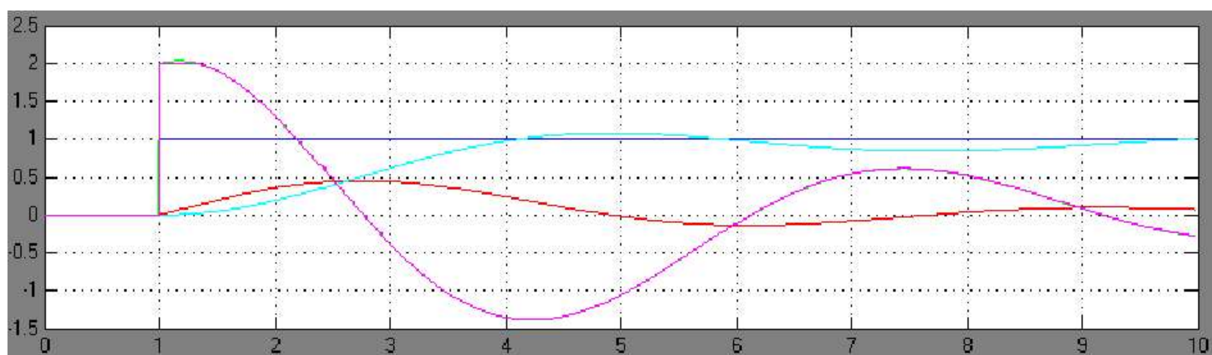


Figure 1.45: Simulation result

- The left side has been zoomed in in the following diagram. Explain the differences between the Matlab controller and the C code controller.

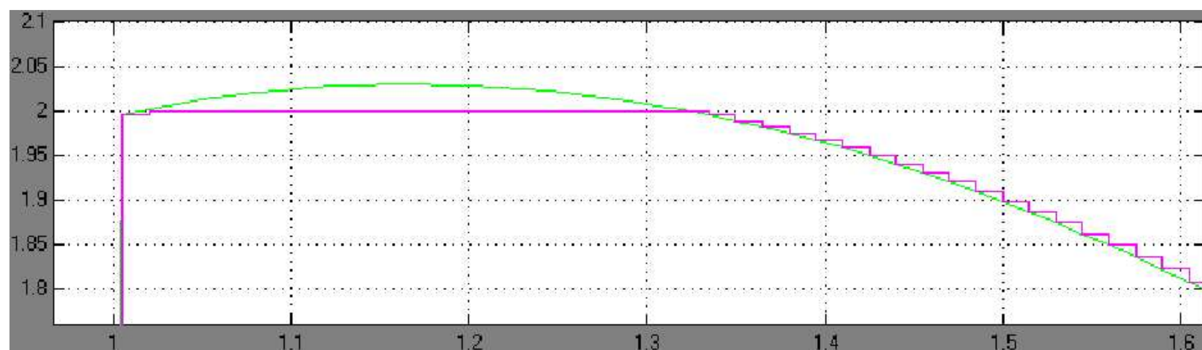


Figure 1.46: Zoomed actuator values (green Matlab, magenta C code)

***

7 Identification of Process Subsystems

System identification becomes easier to handle if only parts of the process are regarded. If the pendulum movement does not change the cart position – which is the case – only the pendulum subsystem can be modeled and identified.

7.1 Subsystem Modelling

For the pendulum subsystem the cart acceleration becomes the input. Outputs are the states of the pendulum: angle α and angular velocity ω.

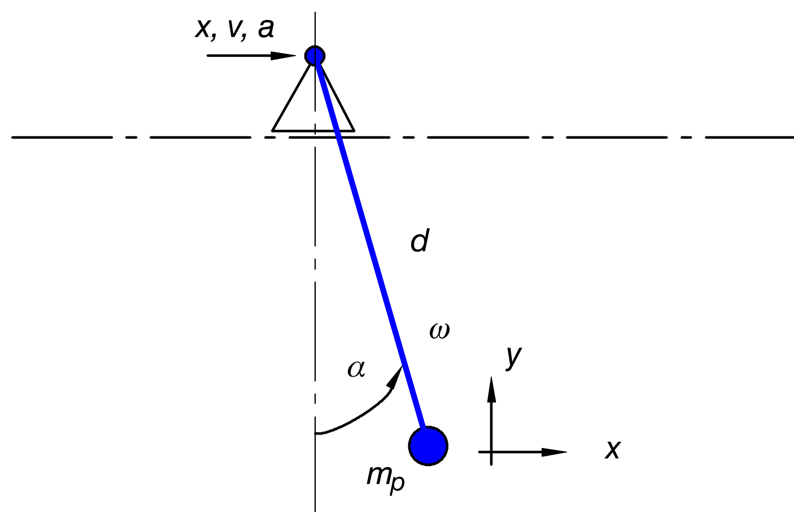


Figure 1.47: Pendulum subsystem

The kinetic energy of the pendulum mass is given by

$$T = \frac{1}{2} m_p v_x^2 + \frac{1}{2} m_p v_y^2 \,. \tag{1.48}$$

Here v_x is the sum

$$v_x = v + \omega d \cos\alpha \,, \tag{1.49}$$

while

$$v_y = \omega d \sin\alpha \,. \tag{1.50}$$

Hence, T becomes

$$T = \frac{1}{2} m_p \left[v^2 + 2v\omega d \cos\alpha + \omega^2 d^2 \left(\cos^2\alpha + \sin^2\alpha \right) \right] , \tag{1.51}$$

$$T = \frac{1}{2} m_p \left(v^2 + 2v\omega d \cos\alpha + \omega^2 d^2 \right) . \tag{1.52}$$

The potential energy depends on the height of m_p

$$V = m_p gh = m_p gd(1 - \cos\alpha) . \tag{1.53}$$

With the Lagrange energy $L = T - V$ the equation of motion becomes

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial\omega} - \frac{\partial L}{\partial\alpha} = 0 . \tag{1.54}$$

With the acceleration $a = \frac{\mathrm{d}v}{\mathrm{d}t}$ the equation (1.54) results in

$$m_p da\cos\alpha - m_p dv\omega\sin\alpha + m_p d^2\frac{\mathrm{d}\omega}{\mathrm{d}t} + m_p dg\sin\alpha = 0 . \tag{1.55}$$

We introduce a friction term, since the pendulum oscillation amplitude decreases over time

$$m_p da\cos\alpha - m_p dv\omega\sin\alpha + m_p d^2\frac{\mathrm{d}\omega}{\mathrm{d}t} + m_p dg\sin\alpha + k_f\omega = 0 . \tag{1.56}$$

Solving for the derivative of ω we obtain

$$\frac{\mathrm{d}\omega}{\mathrm{d}t} = -\frac{a}{d}\cos\alpha + \frac{v\omega}{d}\sin\alpha - \frac{g}{d}\sin\alpha - \frac{k_f}{m_p d^2}\omega , \tag{1.57}$$

or

$$\frac{\mathrm{d}\omega}{\mathrm{d}t} = -\frac{a}{d}\cos\alpha - \frac{g - v\omega}{d}\sin\alpha - \frac{k_f}{m_p d^2}\omega . \tag{1.58}$$

The simulation of the system in fig. 1.48 shows the pendulum oscillation due to an acceleration input.

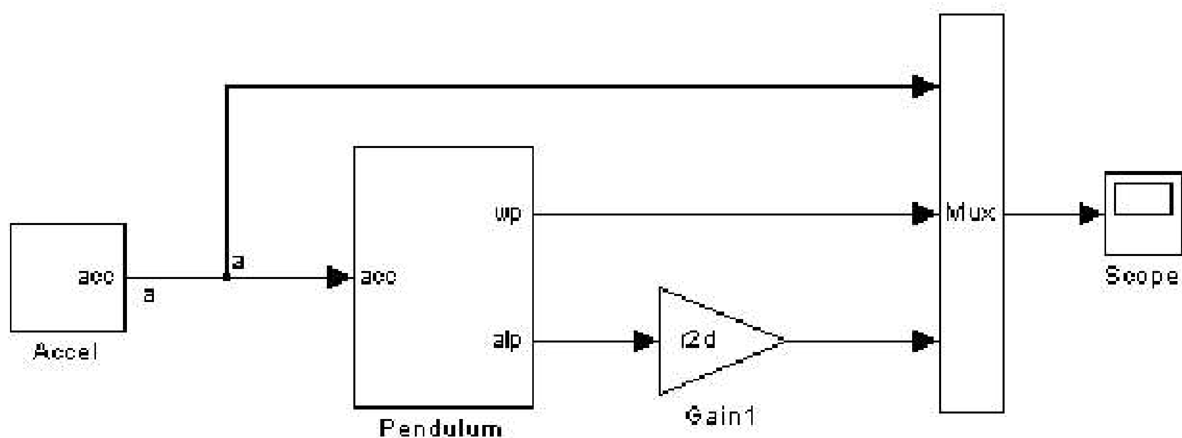


Figure 1.48: Pendulum simulation

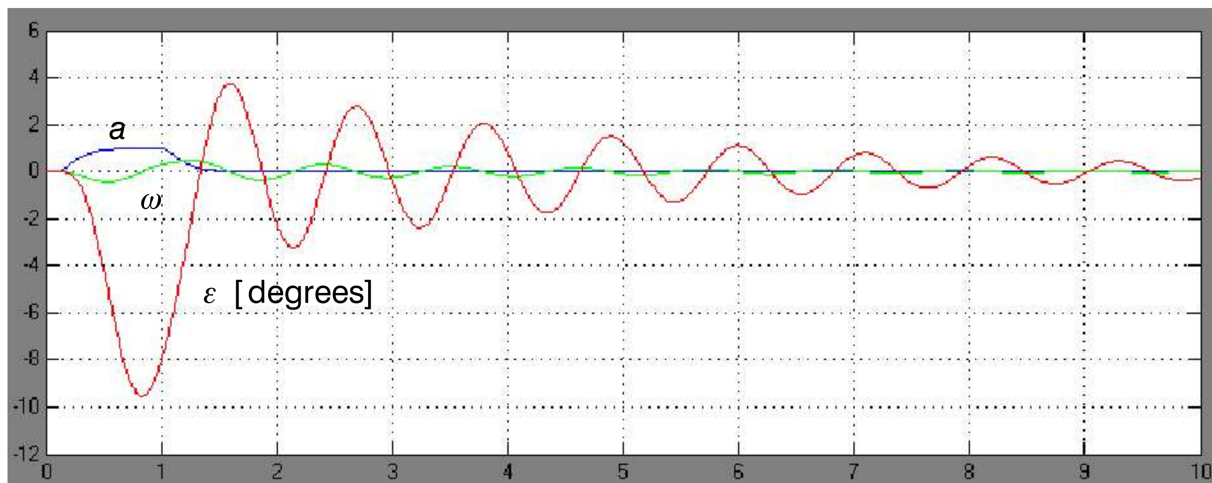


Figure 1.49: Pendulum oscillations

- Write a Simulink simulation for the equation (1.57)
  Define a "PreLoadFcn" to initialize simulation constants:

```
% partialsim_init

d = 0.30;
g = 9.80665;
r2d = 180.0 / pi;
kf = 0.01;   % friction coefficient
mp = 0.2;   % mass of pendulum
kfp = kf / mp / d;
```

- Record process data by moving the cart by motor or by hand to create an acceleration input. Change the simulation parameters to match simulation and process data.

For small angles and small coriolis torque, i.e.

$$\cos\alpha \approx 1 , \quad \sin\alpha \approx \alpha , \quad v\omega \ll g , \tag{1.59}$$

the equation of motion (1.57) becomes linear

$$\frac{\mathrm{d}\omega}{\mathrm{d}t} \approx -\frac{a}{d} - \frac{g}{d}\alpha - \frac{k_f}{m_p d^2}\omega . \tag{1.60}$$

In this case a least squares identification can be used to obtain process parameters with iterative changing process model parameters. The z-transform of (1.60) is required for this purpose.

7.2 ARX Parameter Identification

The ARX form (**AutoRegressive** model with e**X**ternal input) is the simplest structure for parameter identification. The measurement data may contain stochastic noise with mean value of zero. If the measurement data satisfies this condition the parameter estimation will result in the exact parameter values. The structure of the measurement process shows figure 1.50.

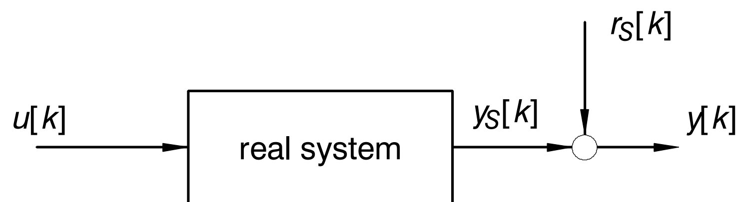


Figure 1.50: Measurement signals

A model should match the transfer characteristic from input u to the measured output y. This becomes possible when the model contains a component to estimate the signal r_S. Several possibilities exist for identification purposes. A general approach is the definition of a transfer function $G_r(z)$.

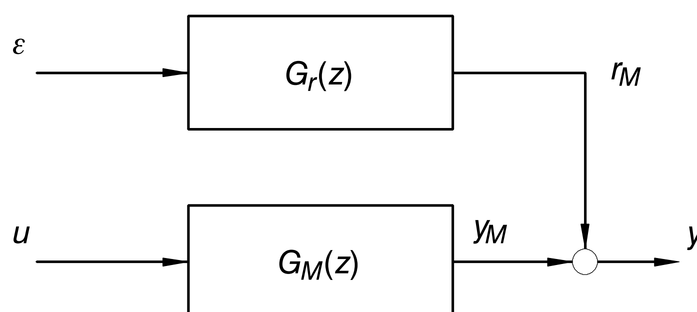


Figure 1.51: Process model G_M and disturbance model G_r

The signal ε is assumed to be a discrete white noise signal (uncorrelated).

The output signal becomes

$$y = y_M + r_M = y_M + G_r \epsilon \,. \tag{1.61}$$

The “model error” follows from (1.61) as

$$\epsilon = G_r^{-1}\left(y - y_M\right) . \tag{1.62}$$

The complete block diagram with the real system explains the building of ε.

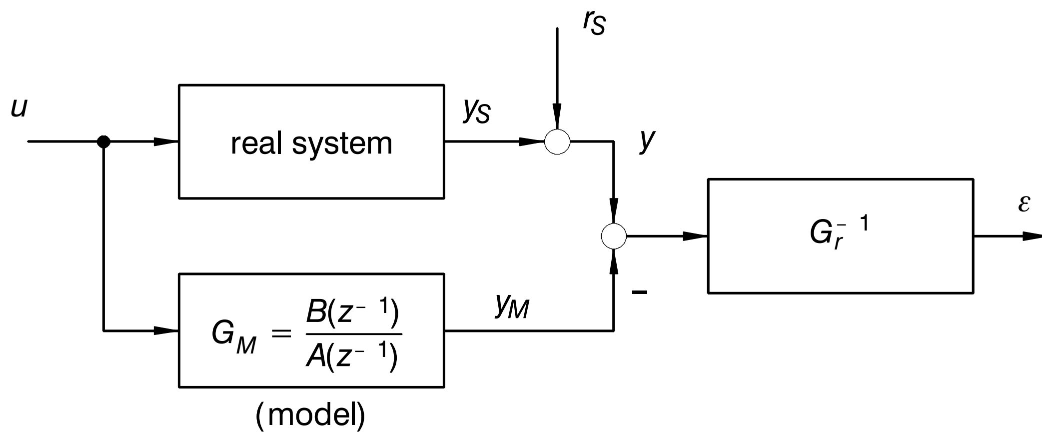


Figure 1.52: Building of the model error ε

It turns out that definitions become easier if G_r is given by

$$G_r(z) = \frac{1}{A(z^{-1})} G_r^*(z) . \tag{1.63}$$

With the definition of G_r^* and the block diagram 1.51 the measurements are

$$y = \frac{B(z^{-1})}{A(z^{-1})} u + \frac{1}{A(z^{-1})} G_r^* \epsilon \tag{1.64}$$

or – by multiplying with A(z-1) – we obtain the well-known form

$$A(z^{-1}) y - B(z^{-1}) u = G_r^* \epsilon . \tag{1.65}$$

From the corresponding block diagram it can be seen what assumptions with respect to G_r^* should be made.

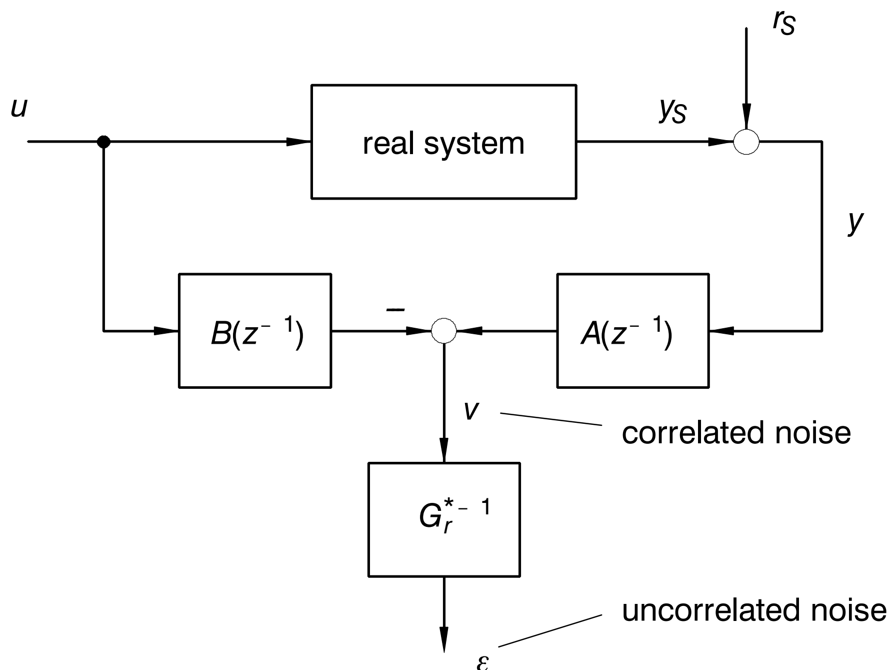


Figure 1.53: Building of the (uncorrelated) model error ε

The structure of the transfer function G_r^* must be suited to describe the effect of creating the correlated noise v from uncorrelated noise ε.

The different parameter identification methods differ in the assumptions made for G_r^*. The most simplest method is ARX which is

$$G_r^* = 1 , \qquad G_r = \frac{1}{A(z^{-1})} . \tag{1.66}$$

In this case we have

$$v[k] = \epsilon[k] = \sum_{\nu=0}^{n} a_\nu y[k - \nu] - \sum_{\nu=1}^{n} b_\nu u[k - \nu] \tag{1.67}$$

for the process model

$$G_M = \frac{\sum_{\nu=1}^{n} b_\nu z^{-\nu}}{\sum_{\nu=0}^{n} a_\nu z^{-\nu}} , \quad a_0 = 1 . \tag{1.68}$$

If G_r^* is not equal to 1 we have more parameters to identify. Solving (1.67) for $y[k]$ we get

$$y[k] = - \sum_{\nu=1}^{n} a_\nu y[k - \nu] + \sum_{\nu=1}^{n} b_\nu u[k - \nu] + \epsilon[k] . \tag{1.69}$$

With the vector of model parameters

$$p = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \\ - \\ b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \quad (1.70)$$

(1.70) can be written as vector product

$$y(k) = m(k)\, p + \epsilon(k)\ . \quad (1.71)$$

The measurement vector is given by

$$m(k) = \left[-\ y[k-1] \ldots -\ y[k-n] \,|\, u[k-1] \ldots u[k-n] \right]. \quad (1.72)$$

If N is the number of measurements the vector equation (1.71) can be extended to a matrix equation. We define the output vector

$$y(N) = \begin{bmatrix} y[n+1] \\ y[n+2] \\ \vdots \\ y[N] \end{bmatrix}, \quad (1.73)$$

the error vector

$$\epsilon(N) = \begin{bmatrix} \epsilon[n+1] \\ \epsilon[n+2] \\ \vdots \\ \epsilon[N] \end{bmatrix}, \quad (1.74)$$

and the data matrix

$$M(N) = \begin{bmatrix} m(n+1) \\ m(n+2) \\ \vdots \\ m(N) \end{bmatrix}. \quad (1.75)$$

All measurement data can now be expressed by the matrix equation

$$y(N) = M(N)\, p + \epsilon(N) \quad (1.76)$$

which allows the calculation of the parameter vector p.

7.2.1 Numerical Solution of the Estimation Problem

The coefficient vector p has $2n$ elements. If the number of measurements N exceeds $3n$ (2n equations), a cost function is required to find an optimal solution for p. A quadratic cost functions leads to the well-known linear equation system (LSE – least squares estimation).

The cost function is (the factor 1/2 makes the solution nice)

$$I = \frac{1}{2} \sum_{k=n+1}^{N} \epsilon^2(k) = \frac{1}{2}\epsilon^T(N)\,\epsilon(N) \Rightarrow \min \;. \tag{1.77}$$

According to (1.76) the error vector is

$$\epsilon(N) = y(N) - M(N)\,p\,. \tag{1.78}$$

Therefore the cost function becomes

$$I = \frac{1}{2}\left[\left(y(N) - M(N)\,p\right)^T \left(y(N) - M(N)\,p\right)\right]. \tag{1.79}$$

The cost function I will be minimal if the first partial derivative by p is zero. The optimal parameter vector is called $\hat{p}$.

$$\left.\frac{\partial I}{\partial p}\right|_{p=\hat{p}} = -\,M^T(N)\,y(N) + M^T(N)\,M(N)\,\hat{p} = 0 \tag{1.80}$$

The optimal solution thus becomes

$$\hat{p} = \left[M^T(N)\,M(N)\right]^{-1} M^T(N)\,y(N)\,. \tag{1.81}$$

The term $(M^T M)^{-1} M$ is called the pseudo-inverse of M. Note that in any real application it it never required to build the inverse of this matrix. Instead only the solution of a set of linear equations is necessary.

An efficient calculation of $M^T M$ can be seen if we take a closer look at the structure of this product.

$$M^T(N)\,M(N) = \begin{bmatrix} -y(n) & -y(N\text{-}1) \\ -y(n\text{-}1) & -y(N\text{-}2) \\ \vdots & \vdots \\ -y(1) & -y(N\text{-}n) \\ \hline u(n) & u(N\text{-}1) \\ u(n\text{-}1) & u(N\text{-}2) \\ \vdots & \vdots \\ u(1) & u(N\text{-}n) \end{bmatrix} \begin{bmatrix} -y(n) & \ldots & -y(1) & u(n) & \ldots & u(1) \\ -y(n+1) & & & u(n+1) & & \\ \vdots & & & \vdots & & \\ -y(N\text{-}1) & \ldots & -y(N\text{-}n) & u(N\text{-}1) & \ldots & u(N\text{-}n) \end{bmatrix}$$

It is an efficient way to compute $M^T M$ by summing the **dyadic** products (shaded column and row – they are identical!). The right product shows the following structure.

$$M^T(N)\, y(N) = \begin{bmatrix} -y(n) & -y(N\text{-}1) \\ -y(n\text{-}1) & -y(N\text{-}2) \\ \vdots & \vdots \\ -y(1) & -y(N\text{-}n) \\ \hline u(n) & u(N\text{-}1) \\ u(n\text{-}1) & u(N\text{-}2) \\ \vdots & \vdots \\ u(1) & u(N\text{-}n) \end{bmatrix} \begin{bmatrix} y(n+1) \\ y(n+2) \\ \vdots \\ y(N) \end{bmatrix}$$

In a similar way the shaded areas can be multiplied.

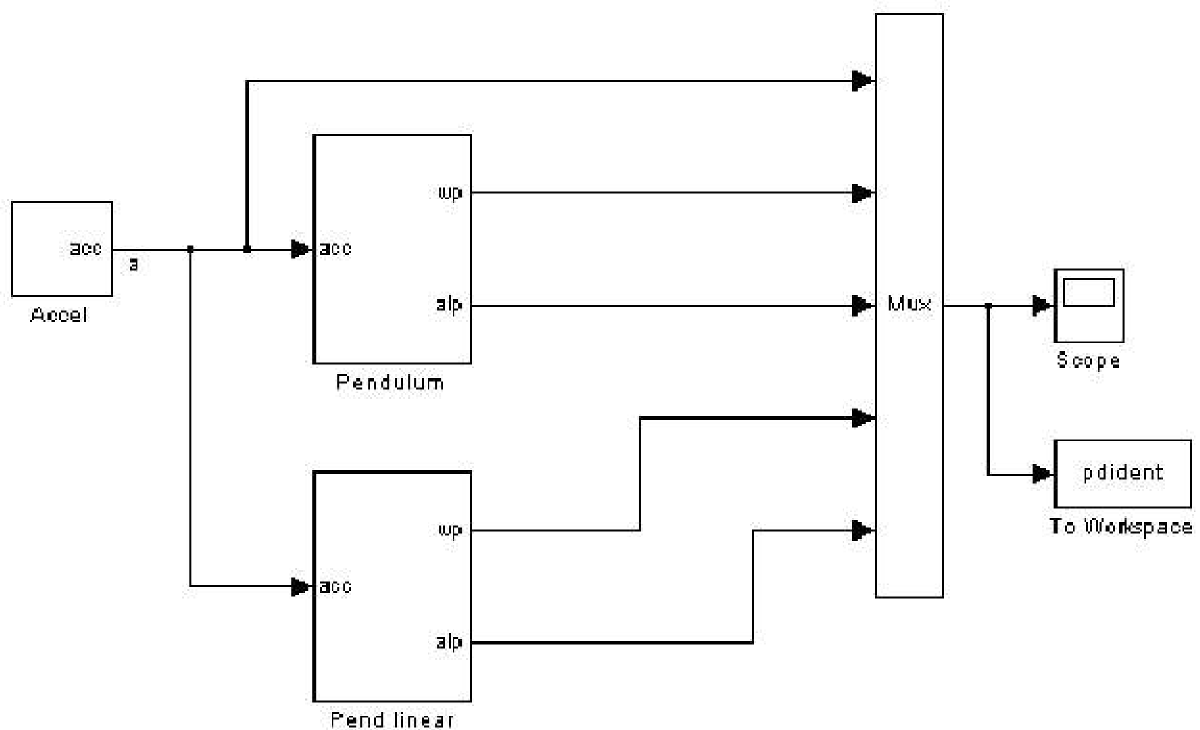
After M^TM and $M^T Y(N)$ are computed, the optimal parameter vector is the solution to the linear set of equations. The following Matlab function illustrates this:

```
function p = lsqid(y, u, n)
% lsqid:  least squares identification
%         p = lsqid(y, u, n)
%
%         where p = [ a1 a2 ... an  b1 b2 ... bn ]

% (c) 2011 Univ. Bremerhaven

N = length(y);
if length(u) ~= N
    error('*** u and y differ in dimenstions.');
end

mk = zeros(2*n, 1);
MTM = zeros(2*n, 2*n);
MTY = mk;
for k=n+1:N
    for h=1:n
        mk(h) = -y(k-h);
        mk(h+n) = u(k-h);
    end;
    MTM = MTM + mk * mk';
    MTY = MTY + mk * y(k);
end
p = MTM \ MTY;
```

Lab #03:

7.3 Identification of the System from Section 7.1

- Extend the simulation of the pendulum system by a linearized system. The simulation should look similar to the next figure.



- Provide the “pident” data output to the Matlab workspace.
- Perform parameter identification based on simulation data from the linear and the nonlinear model data by using the “lsqid” function from section 7.2.1. Verify that the identification was successful.
- Compute physical parameters of the process. This involves some additional theory – see below.

In section 7.1 we derived the linearized ODE

$$\frac{\mathrm{d}\omega}{\mathrm{d}t} = -\frac{a}{d} - \frac{g}{d}\alpha - \frac{k_f}{m_p d^2}\omega \,. \tag{1.82}$$

The differential equation for the pendulum angle follows from (1.82)

$$\frac{d^2\alpha}{dt^2} + \frac{d\alpha}{dt}\frac{k_f}{m_p d^2} + \frac{g}{d}\alpha = -\frac{a}{d} . \tag{1.83}$$

The corresponding transfer function becomes

$$G_\alpha(s) = -\frac{1/d}{s^2 + \frac{k_f}{m_p d^2}s + \frac{g}{d}} . \tag{1.84}$$

The natural frequency and damping factor can be calculated as

$$\omega_0 = \sqrt{\frac{g}{d}} , \qquad D = \frac{1}{2}\sqrt{\frac{d}{g}}\frac{k_f}{m_p d^2} . \tag{1.85}$$

The dc-gain of G_α is $-1/g$ (can only be used as proof of parameter estimation).

On the other hand we have the identified discrete model

$$G_{id}(z) = \frac{b_1 z + b_2}{z^2 + a_1 z + a_2} . \tag{1.86}$$

This transfer function must have the same natural frequency and the same damping. The discrete poles of (1.86) are in s-domain due to

$$z_k = e^{s_k T_s} , \tag{1.87}$$

$$s_k = \frac{\ln(z_k)}{T_s} . \tag{1.88}$$

Therefore

$$\omega_0 = |s_k| \tag{1.89}$$

and

$$D = -\cos\left\{\operatorname{atan}\left\{\frac{\operatorname{Im}\{s_k\}}{\operatorname{Re}\{s_k\}}\right\}\right\} . \tag{1.90}$$

- Calculate g from dc-gain
- Calculate d from ω_0
- Calculate k_f / m_p from D.
- Verify the above results.

***

7.4 Recursive Identification (RLS)

The ARX identification form section 7.2 requires the acquisition of a complete set of data and the solution of a set of equation. A recursive identification does not require the storage of all data and avoids the solution of a matrix equation. The estimated parameters are updated with every new pair $u[k]$, $y[k]$ of data. However, it takes time until the estimated parameters converge to their exact values. Recursive estimation allows to track (slow) changes of model parameters over time.

As we have seen that the update of the matrix product $M^T M = M^*$ can be expressed as

$$M^*(k+1) = M^*(k) + m(k+1)\, m^T(k+1) \tag{1.91}$$

(dyadic product). With

$$p[k] = M^{*-1} M^* y(k) \tag{1.92}$$

the next result (if a new pair of data was available) is

$$p[k+1] = M^{*-1}(k+1)\left(M^*(k)\, p[k] + m(k+1)\, y[k+1]\right). \tag{1.93}$$

Solving (1.91) for $M^*(k)$ and substituting this in (1.93) gives an expression how $p[k+1]$ can be computed from $p[k]$. We will not perform some longer computation with this expressions but it it turn out, that the update of p can be done without solving a set of equations. The drawback with this is that we need initial guesses of parameters. If these parameters are not correct they will converge to the same values as ARX least squares parameter estimation.

The *prediction error* of the new data is

$$\epsilon[k+1] = y[k+1] - m^T(k+1)\, p(k). \tag{1.94}$$

With the prediction error the parameter vector can be improved by

$$p[k+1] = p[k] + q(k+1)\, \epsilon[k+1]. \tag{1.95}$$

Here $q(k+1)$ is given by

$$q(k+1) = \frac{P(k)\, m(k+1)}{1 + m^T(k+1)\, P(k)\, m(k+1)}. \tag{1.96}$$

Note the denominator is a scalar so a simple division is required instead of a matrix inversion. $P(k)$ is the so-called *error covariance* matrix which can be updated according to

$$P(k+1) = P(k) - q(k+1)\, m^T(k+1)\, P(k). \tag{1.97}$$

The equations (1.94) to (1.97) describe the recursive least squares estimation algorithm.

It becomes obvious that initial guesses for $p(n)$ [the model parameters] and $P(n)$ [the error covariance matrix] are necessary. A possible choice for $p(n)$ is $p(n) = 0$. The error

covariance matrix $P(n)$ (dimension is 2n x 2n) must not by zero (why?) and can be set to a diagonal matrix with small values (e.g.. 0.1 x unity matrix). Good initial values improves the convergence to the exact values.

7.4.1 Extended Model Identification (Disturbance Model)

In all practical identification applications the assumption that $G_r^* = 1$ is not satisfied. The estimation of unbiased coefficients therefore requires the identification of a disturbance model

$$G_r^* = \frac{C(z^{-1})}{D(z^{-1})}. \tag{1.98}$$

With RLS this can be easily done by extending the parameter vector with the coefficients of the disturbance model

$$p = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \\ \hline b_1 \\ b_2 \\ \vdots \\ b_n \\ \hline c_1 \\ c_2 \\ \vdots \\ c_n \\ \hline d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} \tag{1.99}$$

The vector of “measurements” need to changed according to

$$\begin{aligned} m(k) = [-y[k-1] \ldots -y[k-n] \mid u[k-1] \ldots u[k-n] \\ \epsilon[k-1] \ldots \epsilon[k-n] \mid -v[k-1] \ldots -v[k-n]]. \end{aligned} \tag{1.100}$$

Of course, ε and v are unknown. They will be replaced by the estimates

$$\hat{\epsilon}[k] = y[k] - m(k)\, p[k-1], \tag{1.101}$$

$$\hat{v}[k] = y[k] - m_{yu}(k)\, p_{ab}[k-1], \tag{1.102}$$

where m_{yu} consists of only y and u [subvector of $m(k)$] and p_{ab} is the upper part of p (containing only a_k and b_k coefficients).

The equations (1.95) to (1.97) for RLS are still valid for extended RLS. However, the dimensions change to

$$m(k) \in \Re^{4nx1}, \quad P(k) \in \Re^{4nx4n}. \tag{1.103}$$

Example:
The process

$$G(z) = \frac{0.3z - 0.2}{z^2 - 1.8z + 0.9} \tag{1.104}$$

could be identified as

$$G_{id}(z) = \frac{0.26883z - 0.16318}{z^2 - 1.78665z + 0.89110} \tag{1.105}$$

after a large number of iterations. The (correlated) disturbance has a similar magnitude as the input signal u. The step responses of both transfer functions is shown below.

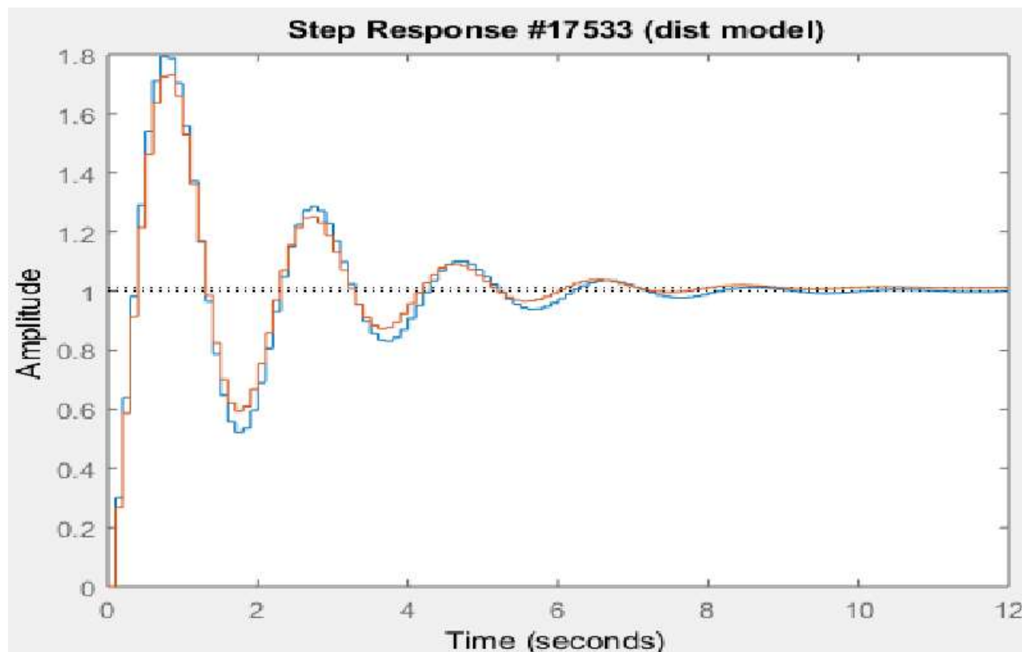


Figure 1.54: Step responses of process G and process model G_{id}

Counterexample:
The identification of the same process *without disturbance model* fails completely due to the systematic errors associated with correlated noise.

The identified process

$$G_{id}(z) = \frac{0.06934z + 0.51655}{z^2 - 0.27968z - 0.18526} \tag{1.106}$$

is unusable.

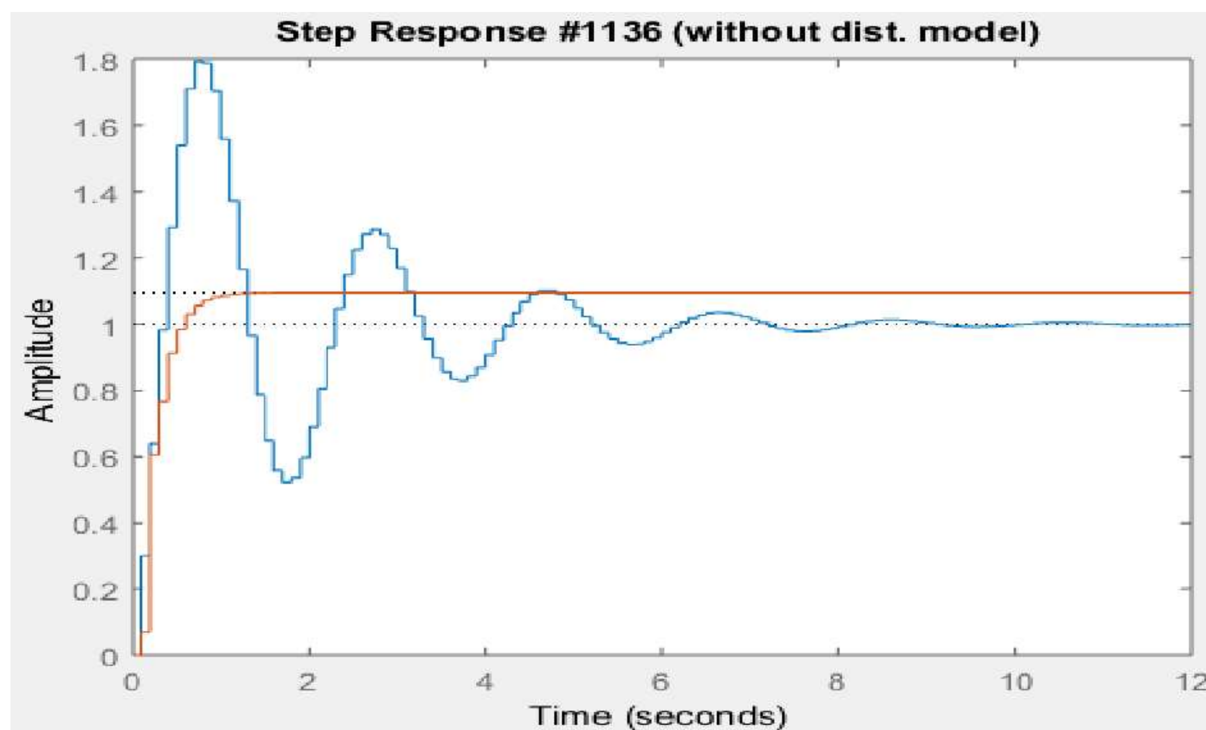


Figure 1.55: Step responses of process G and process model G_{id} without disturbance model (unusable identification result)

Lab #04:

7.5 Recursive Identification of the System from Section 7.1

- Write a Matlab m-file for recursive estimation of the pendulum model. Provide initial values for $p(n)$ and $P(n)$.
- The required sequence of equations is (1.96) → (1.97) → (1.94) → (1.95).
- Repeat the recursive estimation with the same set of data without initializing $p(n)$ and $P(n)$ again. Usually it take a lot of iterations until the model parameters are correct. This is the drawback of recursive identification. The number of iterations depend on the “information content” of available data.
- Verify proper operation by comparing your result with the result from ARX estimation.

***

8 Bibliography

[1] Ashenden, Peter J.: The Designer's Guide to VHDL, 3rd. Ed.
Morgan Kaufmann, 2008

[2] Al-Hashimi.Bashir M.: System-on-Chip: Next Generation Electronics.
Institution of Electrical Engineers, 2006

[3] Below, Klaus and Dietrich, Karin: Medizinische Gerätetechnik (in German).
Europa-Lehrmittel, 2006

[4] Jennings, D., Fint, A., Turton, BCH. and Nokes, LDM: Introduction to Medical Electronics Applications.
Edward Arnold PLC, 1995

[5] Lipsett, Rger, Schaefer, Carls and Ussery Cary: VHDL Hardware Description and Design.
Kluwer Academic 1990

[6] Lyons, Richard G.: Understanding Digital Signal Processing.
Prentice Hall, 2011

[7] Predroni, Volnei A.: Circuit Design and Simulation with VHDL, 2nd. Ed.
MIT Press, 2010

[8] Prutchi, David and Norris, Michael: Design and Development of Medical Electronic Instrumentation.
John Wiley& Sons, 2005

[9] Reis, Ricardo, Lubaszewski, Marcelo and Jess, Jochen: Design of Systems on a Chip: Design and Test.
Springer, 2010

[10] Reichardt, J. und Schwarz, B.: VHDL-Synthese.
Oldenbourg, 2001

[11] Sass, Ron and Schmidt, Andrew G.: Embedded Systems Design with Platform FPGAs: Principles and Practices.
Elsevier Inc. 2010

[12] Wakerly, John F. : Digital Design, Principles & Practices.
Prentice Hall, 2001

***